Malware-Analyse und Reverse Engineering

4: Startup

6.4.2017

Prof. Dr. Michael Engel

Überblick (heute)

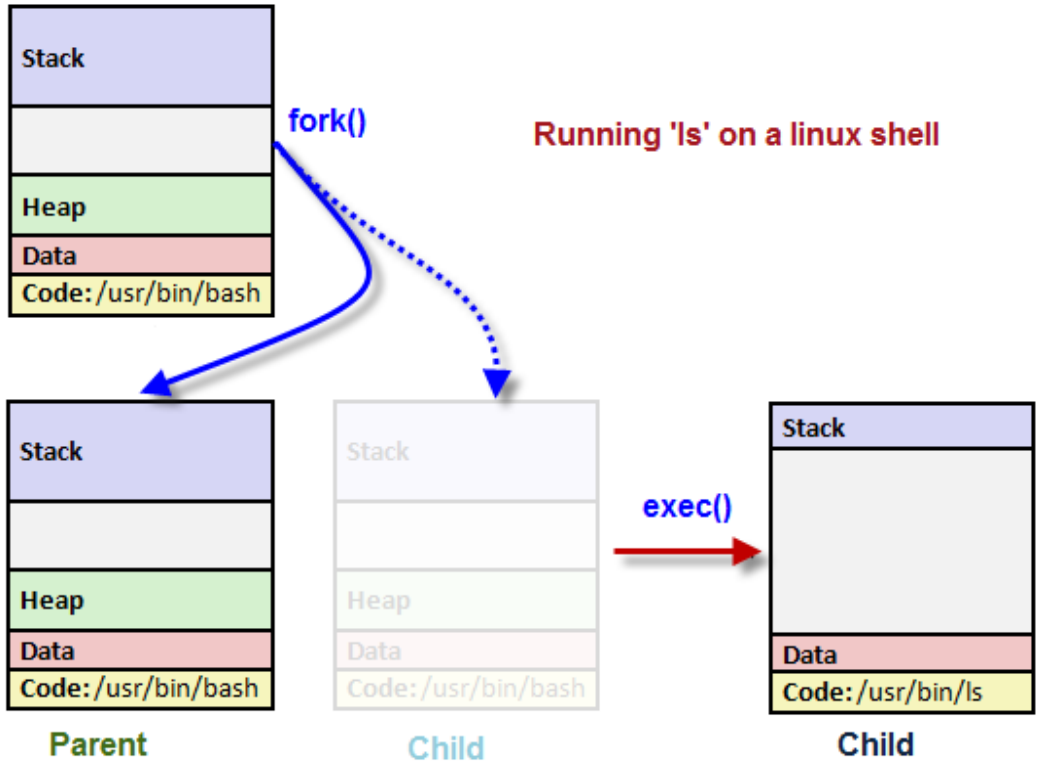
Themen:

- Prozesserzeugung in Linux: fork und exec
- Linker und Einsprungpunkt
- Funktionsaufrufe und Parameterübergabe
- Stackframes

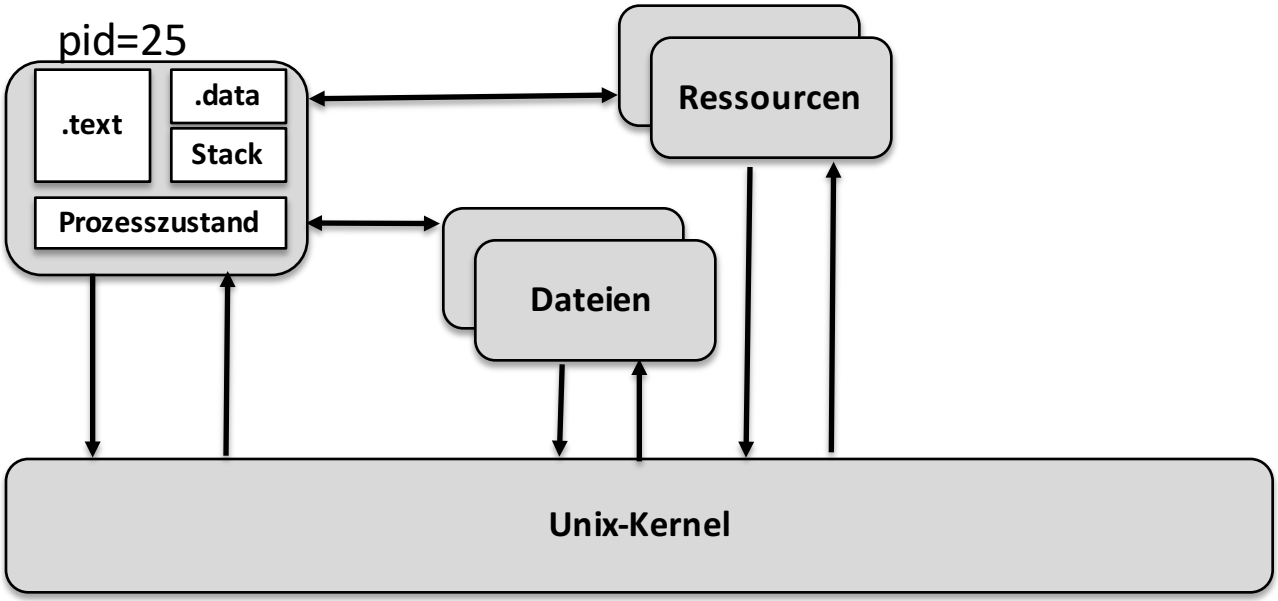
Prozesserzeugung in Linux

Zweistufiger Ablauf:

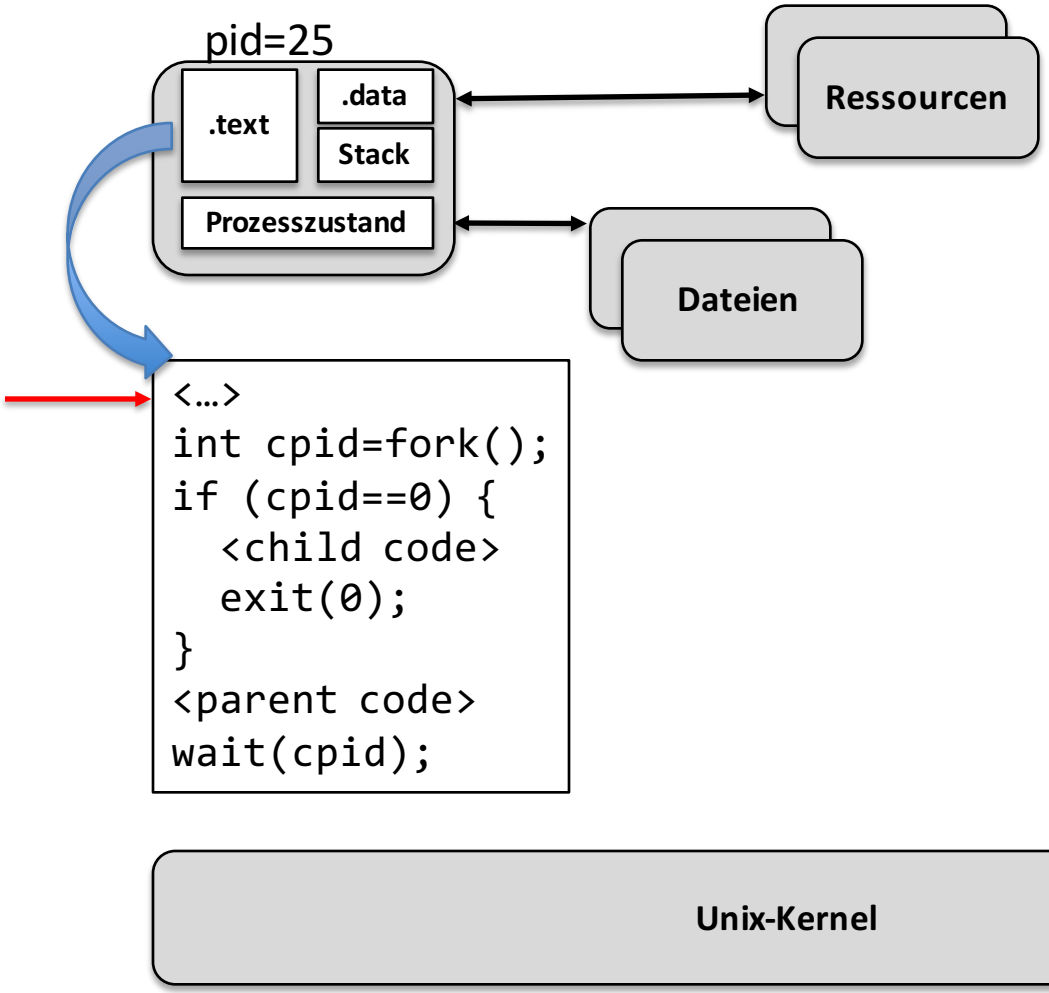
- Prozess erzeugt identische Kopie („Kindprozess“) von sich selbst
  - Systemaufruf `fork()`
  - Erzeugt Kopie des Adressraums
- Kindprozess überschreibt eigenen Adressraum mit Code & Daten eines anderen Programms:
  - Systemaufruf `exec()`



Ablauf von fork (1)

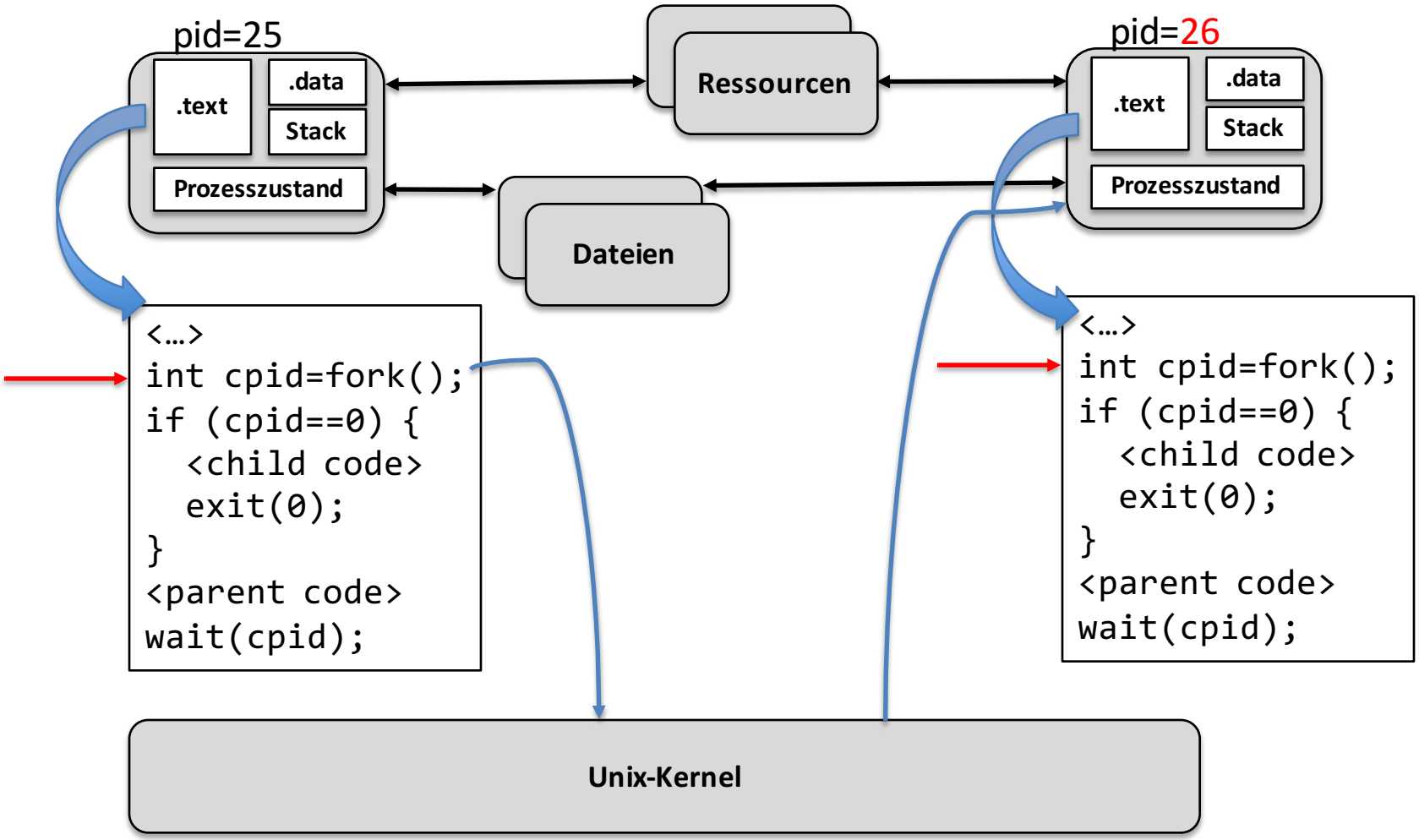


Ablauf von fork (2)



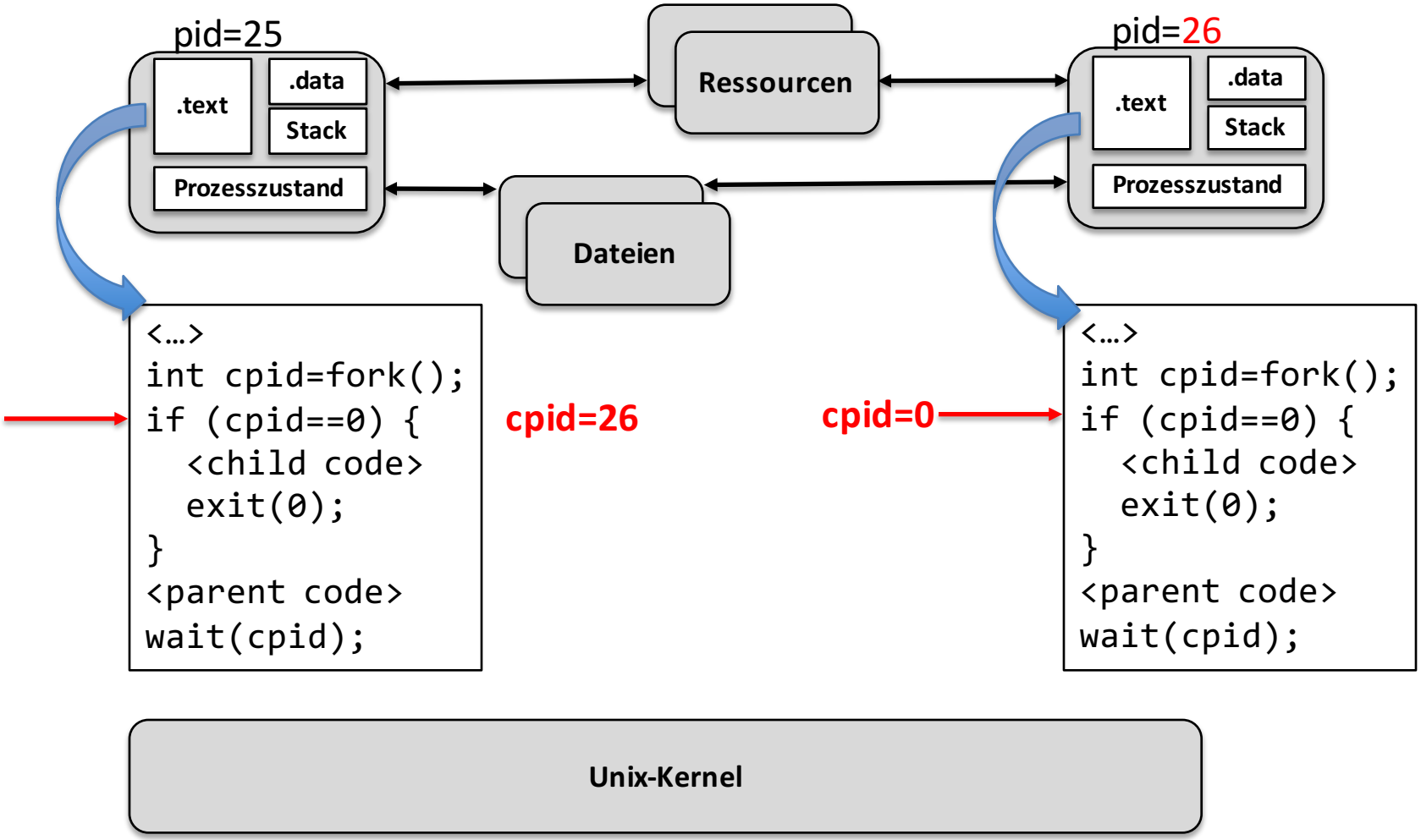
Ablauf von fork (3)

pid26 ist Kopie von Prozess mit pid25, beide in fork()!



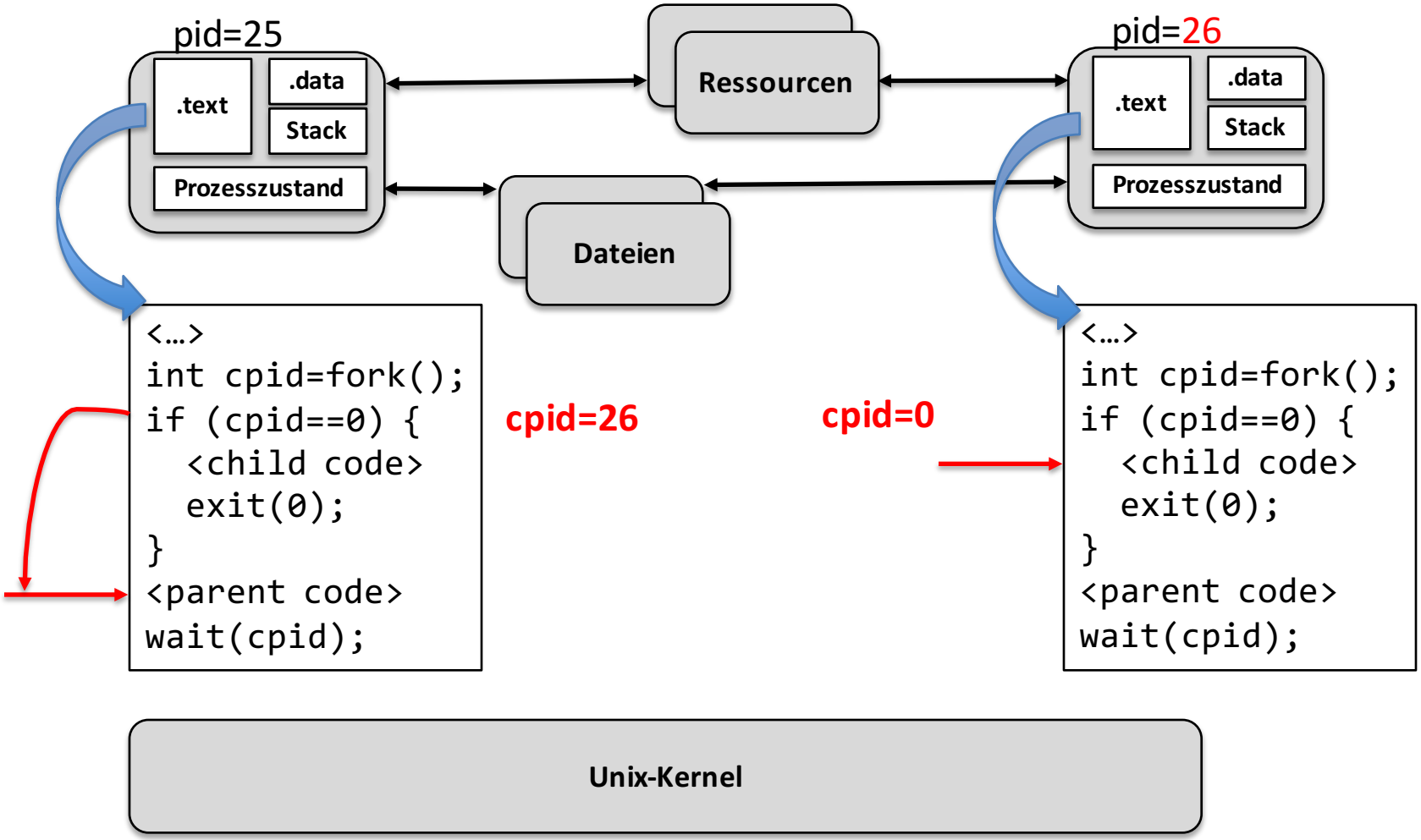
Ablauf von fork (4)

pid25 und pid26 kehren aus fork zurück.
Unterschied: Rückgabewert!



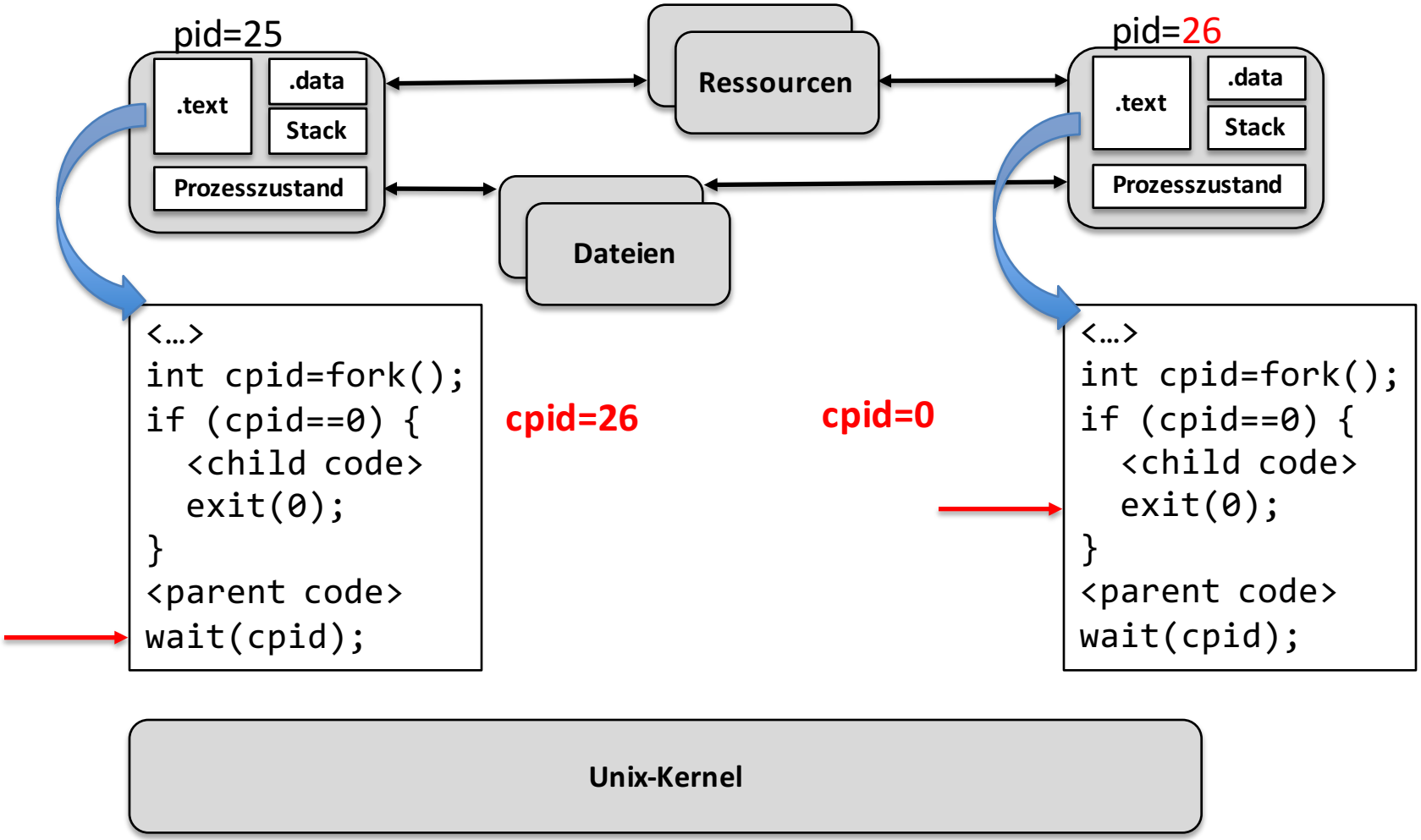
Ablauf von fork (5)

pid25 überspringt if()-Block, pid26 führt ihn aus



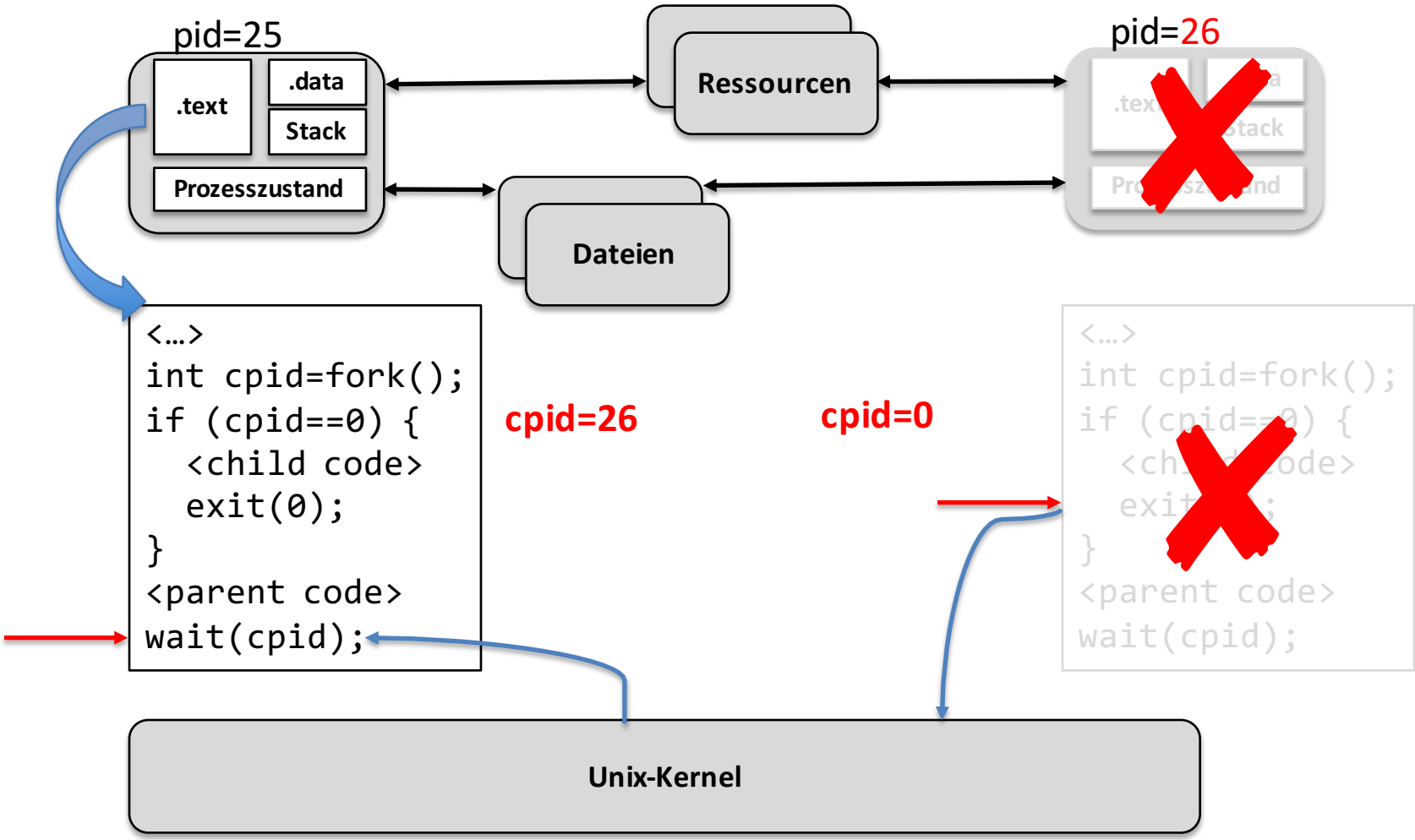
Ablauf von fork (6)

pid25 wartet auf Ende von pid26,
pid26 führt exit(0) aus und beendet sich



Ablauf von fork (7)

**pid26 ist beendet und aus Speicher entfernt,
pid25 beendet sein Warten auf Ende von pid26**



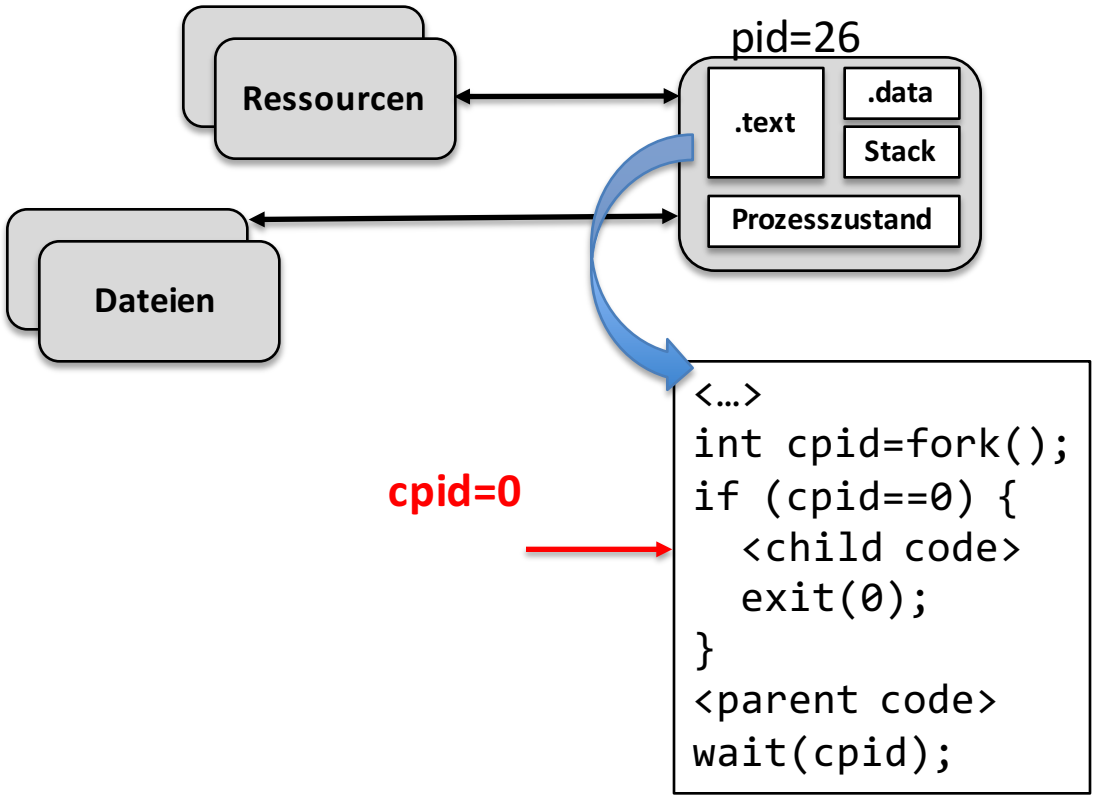
Funktion von exec()

Ersetzen des Speicherinhaltes des aktuellen Prozesses durch die Sektionen einer anderen ausführbaren Datei (Parameter von exec)

- Einzige Methode, *neue* Programme zu starten
  - ...da *fork()* nur Kopie des aufrufenden Prozesses erzeugt
- Kernel öffnet ausführbare Programmdatei
- ELF-Lader lädt Text- und Datensektionen
- ld.so lädt evtl. shared libraries
- Stack, Heap, BSS werden neu angelegt
- Neues Programm wird am *Einsprungpunkt* gestartet

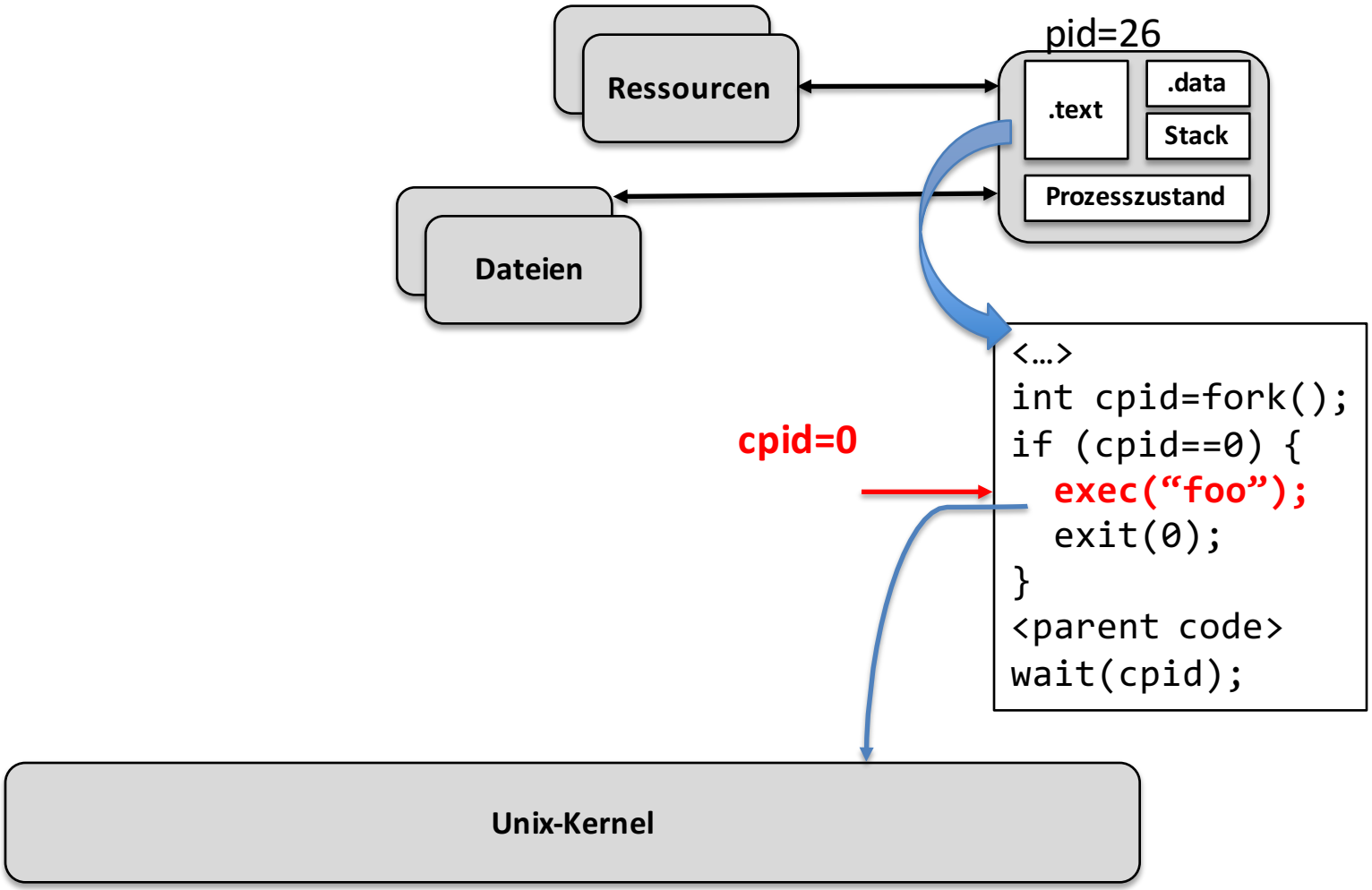
Ablauf von exec (1)

Kindprozess ist in Ausführung des if()-Blocks



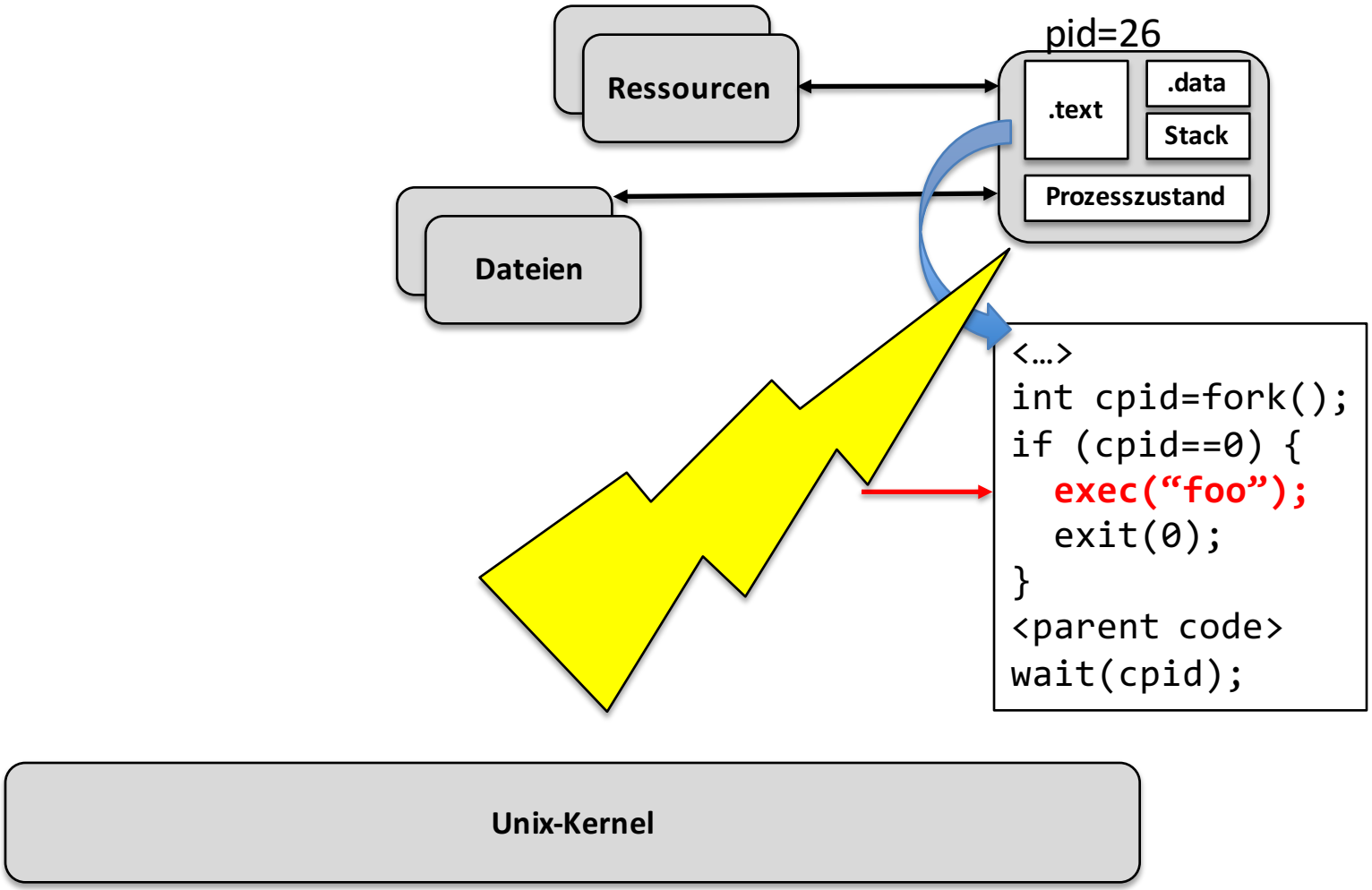
Ablauf von exec (2)

In if-Block wird Systemaufruf exec() aufgerufen



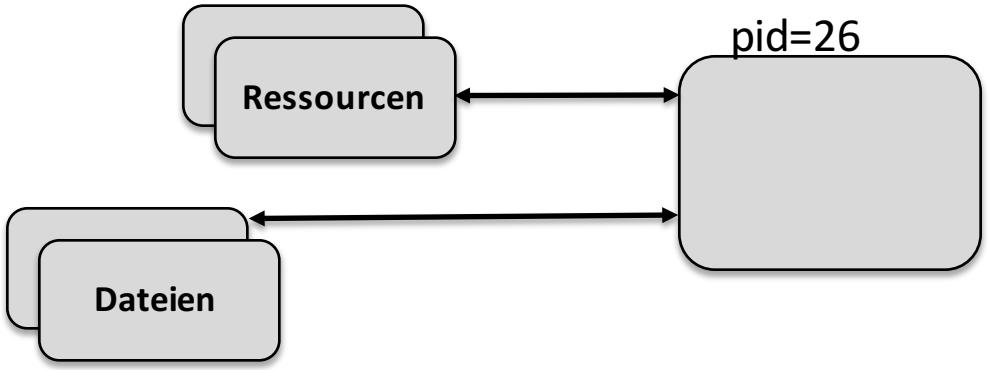
Ablauf von exec (3)

Kernel "entsorgt" Speicherinhalt von pid26



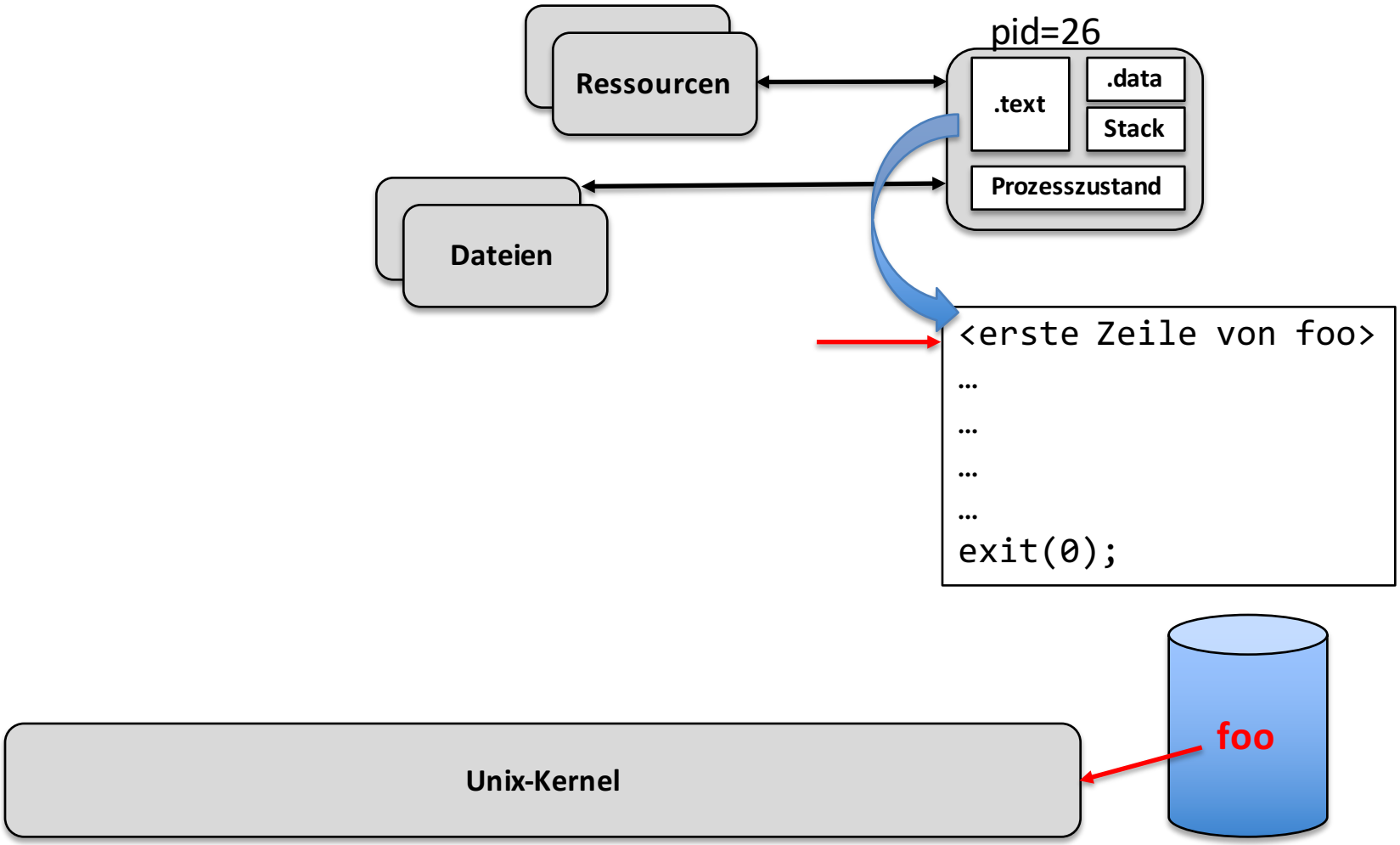
Ablauf von exec (4)

Kernel öffnet und lädt ausführbare Datei “foo” von Dateisystem



Ablauf von exec (5)

Speicher von pid26 durch Sektionen von "foo" ersetzt, startet von "vorne"



Starten des neuen Programms nach exec()

HOCHSCHULE COBURG

Wo startet mit *fork()* neu erzeugter Prozess?

- Ist identische Kopie des Elternprozesses
- Fortsetzung nach Rückkehr aus *fork()*-Aufruf

Wo startet ein neu geladenes Programm?

- Kein alter Programmcode vorhanden
- Start „von vorne“...
- ...also in main()? ➞ ***nein!***

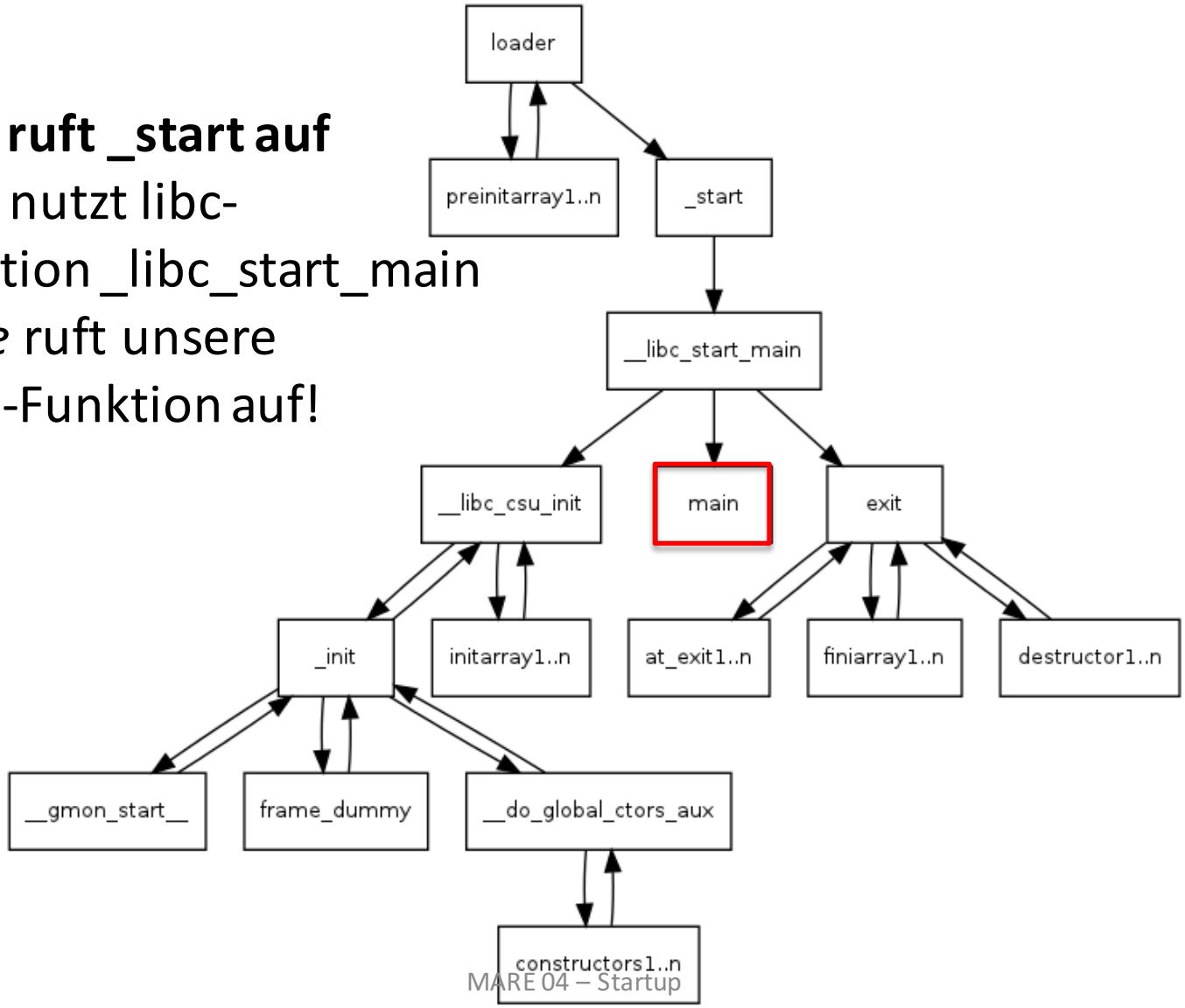
Vor Aufruf von main sind Initialisierungen durchzuführen

- z.B. Nullsetzen von Inhalten der .bss-Sektion

Überblick Programmstart

Loader ruft _start auf

- Start nutzt libc-Funktion _libc_start_main
- *Diese* ruft unsere main-Funktion auf!



Das einfachste C-Programm

...ein leeres main()

- Im Disassembler (Ausschnitte):

```
$ objdump -d prog1
080482e0 <_start>:
80482e0:  31 ed           xor    %ebp,%ebp
80482e2:  5e              pop    %esi
80482e3:  89 e1           mov    %esp,%ecx
80482e5:  83 e4 f0        and    $0xfffffff0,%esp
80482e8:  50              push   %eax
80482e9:  54              push   %esp
80482ea:  52              push   %edx
80482eb:  68 00 84 04 08  push   $0x8048400
80482f0:  68 a0 83 04 08  push   $0x80483a0
80482f5:  51              push   %ecx
80482f6:  56              push   %esi
80482f7:  68 94 83 04 08  push   $0x8048394
80482fc:  e8 c3 ff ff ff  call   80482c4 <__libc_start_main@plt>
8048301:  f4              hlt
```

Adressen der Instruktionen

Opcodes

Disassemblierte Mnemonics und Parameter

prog1.c:

```
int main(void) {
}
```

Übersetzen mit:

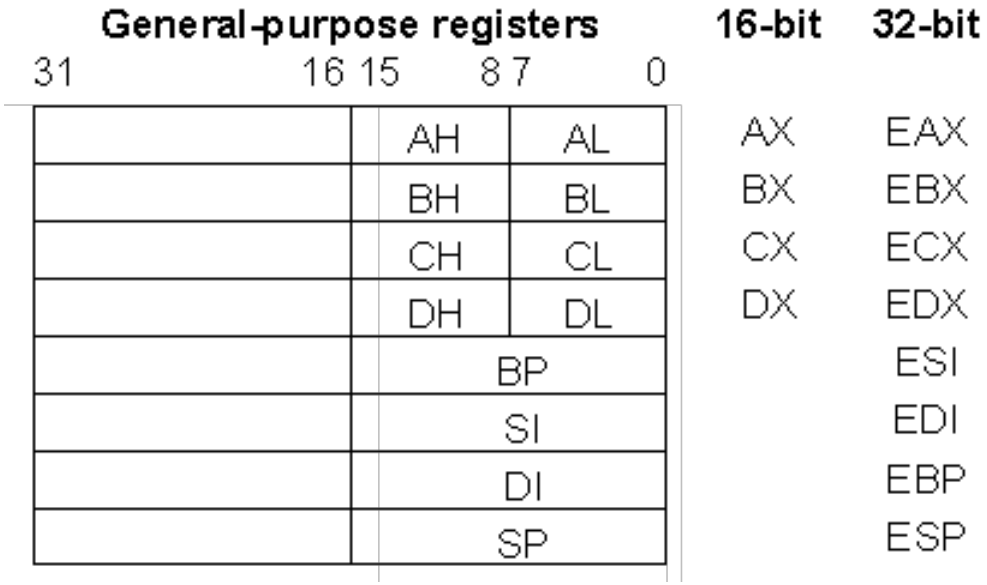
```
gcc -g -o prog1 prog1.c
```

Compiler-Option "-g" fügt zusätzliche Debug-Symbole in ELF-Daten ein

x86-Architektur

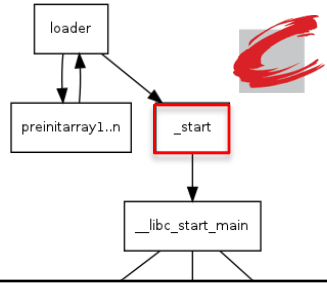
Hier: 32-Bit-Architektur („ia32“)

- Register EAX, EBX, ECX, EDX sind 32 Bit breit (general purpose*)
  - Auch als 16-Bit-Register AX, BX, CX, DX ansprechbar
  - High-/lowbyte von AX, ..., DX als AH/AL, ..., DH/DL ansprechbar
- Zusätzliche Register
  - Stackpointer ESP
  - Basepointer EBP
  - Source- und Destination-Index
    - z.B. für Kopier-befehle



*fast… nicht alle Maschinenbefehle können mit allen 4 Registern arbeiten

Die Funktion start()



HOCHSCHULE COBURG

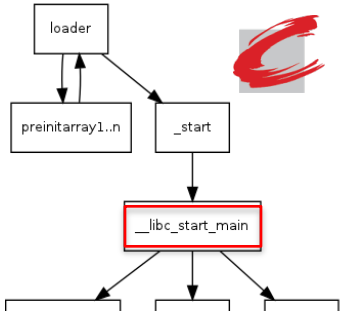
```
080482e0 <_start>:
80482e0: xor    %ebp,%ebp        ; Register ebp auf 0 setzen
80482e2: pop    %esi             ; Register esi von Stack holen
80482e3: mov    %esp,%ecx        ; Stackpointer aus Register ecx kopieren
80482e5: and    $0xfffffff0,%esp ; Stack alignment auf Vielfaches von 16 Byte
80482e8: push   %eax             ; Register eax, esp, edx auf Stack sichern
80482e9: push   %esp
80482ea: push   %edx
80482eb: push   $0x8048400       ; Adresse 0x8048400 auf Stack sichern
80482f0: push   $0x80483a0       ; Adresse 0x80483a0 auf Stack sichern
80482f5: push   %ecx             ; Register ecx, esi auf Stack sichern
80482f6: push   %esi
80482f7: push   $0x8048394       ; Adresse 0x8048394 auf Stack sichern
80482fc: call   80482c4 <__libc_start_main@plt> ; _libc_start_main aufrufen
8048301: hlt                     ; Prozessor anhalten
```

1
2
3

Drei Schritte:

1. Stack-Initialisierung
2. Parameter für _libc_start_main auf Stack & Aufruf
3. Prozessor anhalten

Parameterübergabe an _libc_start_main



HOCHSCHULE COBURG

Prototyp von _libc_start_main:

```
int __libc_start_main(   int (*main) (int, char **, char **),
                         int argc,
                         char ** ubp_av,
                         void (*init) (void),
                         void (*fini) (void),
                         void (*rtld_fini) (void),
                         void (*stack_end)
);
```

Funktionszeiger auf unser main

Argumentzähler argc und

Array von Argumentstrings argv

Funktionszeiger auf Konstruktur und Destruktur für dieses Programm*

Funktionszeiger für Destruktor des dyn. Linkers (Entfernen von shared libraries)

Stackpointer

* Teil der GNU libc-Source in glibc-Datei `csu/libc-start.c`

Die main-Funktion eines C-Programms

Prototyp für main:

```
int main(int argc, char **argv, char **envp);
```

argc: Anzahl der Argumente auf der Befehlszeile (>= 1)

argv: Array mit Text der Argumente (nullterminierte Strings)
argv[0] = Name des aufgerufenenProgramms

envp: Array mit Text der Umgebungsvariablen der Shell (nullterminierte Strings, Form A=B)

```
$ export VAR="fasel"
$ ./prog 1 foo bla

⇒ argc = 4
⇒ argv = { "./prog", "1", "foo",
           "bla", 0 }
⇒ envp = { "VAR=fasel", …, 0 }
```

Environment ausgeben:

```
#include <stdio.h>

int main(int argc, char **argv, char **envp)
{
    while(*envp)
        printf("%s\n",*envp++);
}
```

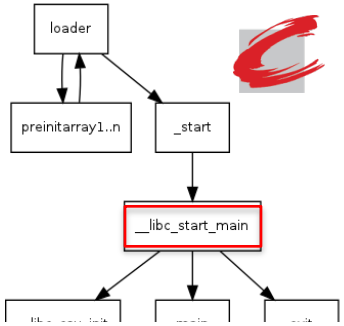
Parameterübergabe an _libc_start_main: Stack

Parameter werden auf Stack *in umgekehrter Reihenfolge* der Signatur übergeben:

```
080482e0 <_start>:
80482e0: xor    %ebp,%ebp
80482e2: pop    %esi
80482e3: mov    %esp,%ecx
80482e5: and    $0xfffffff0,%esp
80482e8: push   %eax       ; unbenutzt
80482e9: push   %esp       ; Stack pointer
80482ea: push   %edx       ; Adr. Destr. ld.so
80482eb: push   $0x8048400 ; Adr. _libc_csu_fini
80482f0: push   $0x80483a0 ; Adr. _libc_csu_init
80482f5: push   %ecx       ; argv
80482f6: push   %esi       ; argc
80482f7: push   $0x8048394 ; Adresse von main
80482fc: call   80482c4 <__libc_start_main@plt>
8048301: hlt
```

value	__libc_start_main arg	content
$eax	Don't know.	Don't care.
%esp	void (*stack_end)	Our aligned stack pointer.
%edx	void (*rtld_fini)(void)	Destructor of dynamic linker from loader passed in %edx. Registered by __libc_start_main with __cxat_exit() to call the FINI for dynamic libraries that got loaded before us.
0x8048400	void (*fini)(void)	__libc_csu_fini - Destructor of this program. Registered by __libc_start_main with __cxat_exit().
0x80483a0	void (*init)(void)	__libc_csu_init, Constructor of this program. Called by __libc_start_main before main.
%ecx	char **ubp_av	argv off of the stack.
%esi	arcg	argc off of the stack.
0x8048394	int(*main)(int, char**,char**)	main of our program called by __libc_start_main. Return value of main is passed to exit() which terminates our program.

Funktion von _libc_start_main



HOCHSCHULE COBURG

Linken und Laden

- Takes care of some security problems with setuid setgid programs
- Starts up threading
- Registers the fini (our program), and rtld_fini (run-time loader) arguments to get run by at_exit to run the program's and the loader's cleanup routines
- Calls the init argument
- Calls the main with the argc and argv arguments passed to it and with the global __environ argument as detailed above
- Calls exit with the return value of main

C-Programmkonstruktor _libc_csu_init

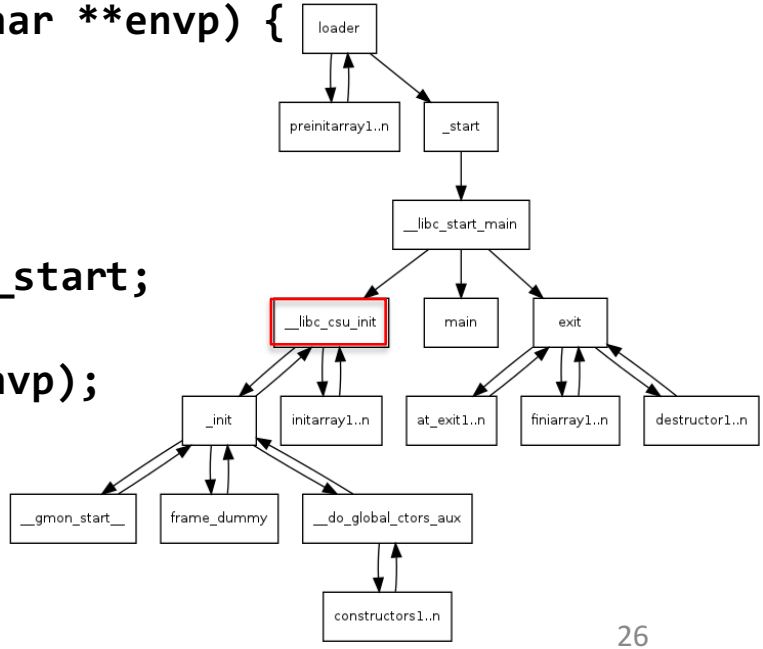
Das ist doch kein C++?!?

- Auch jedes C-Programm hat Funktionen zum Initialisieren und Entfernen von Datenelementen (aber keine Objekte)
  - Konstruktor `__libc_csu_init`, Destruktor `__libc_csu_fini`

```
void __libc_csu_init (int argc, char **argv, char **envp) {

  _init ();

  const size_t size =
                __init_array_end - __init_array_start;
  for (size_t i = 0; i < size; i++)
      (*__init_array_start [i]) (argc, argv, envp);
}
```



Was macht _init?

* in glibc-Source `csu/elf-init.c`

Ausgabe:

```
$ ./prog2
a_constructor
main
```

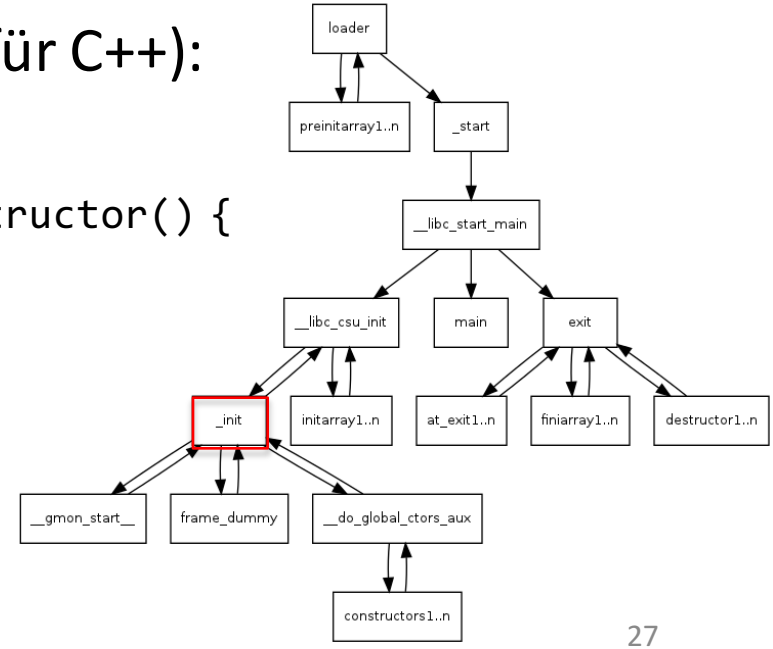
Vorbereitung globaler Informationen

- Zeiger auf globale Tabellen
- Optionaler Start des Profilers
- Speichern von Informationen zur Unterstützung von Exception Handlers
- Aufruf globaler Konstruktoren (auch für C++):

```
#include <stdio.h>
void __attribute__ ((constructor)) a_constructor() {
    printf("%s\n", __FUNCTION__);
}

int main() {
    printf("%s\n", __FUNCTION__);
}
```

gcc-Erweiterung:
Name der aktuellen Funktion

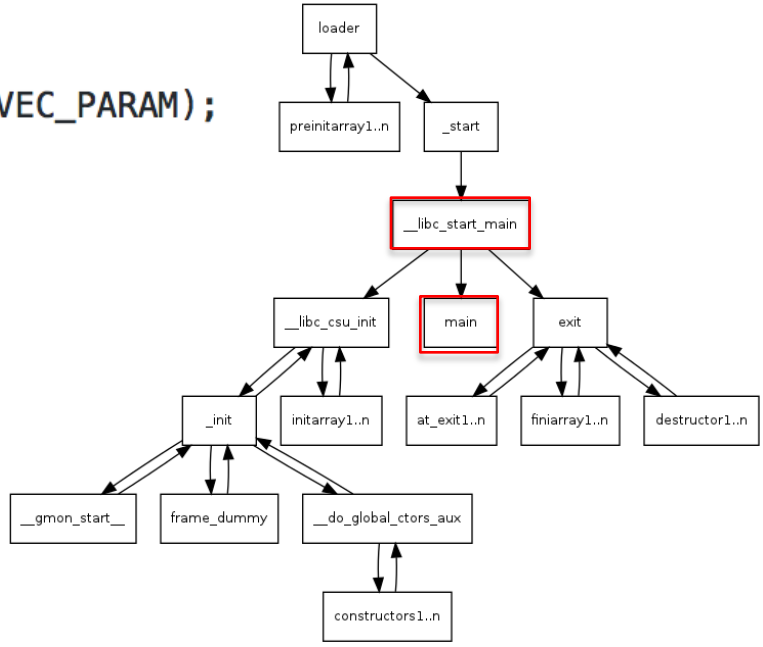


...endlich!

Aufruf von unserem main ist ganz unspektakulär

- Verwendet übergebenen Zeiger auf Main-Funktion
- Übergibt auf Stack erhaltene argc, argv und envp an main

```
253     /* Nothing fancy, just call the function.  */
254     result = main (argc, argv, __environ MAIN_AUXVEC_PARAM);
```



* Teil der GNU libc-Source in glibc-Datei `csu/libc-start.c`

Funktionsaufrufe und der Stack

HOCHSCHULE COBURG

Der Stack verwaltet Informationen über Funktionsaufrufe

- Aufrufhierarchie:
  - Rücksprungadressen
  - Parameter von Funktionen
  - lokale Variable (Gültigkeitsbereich = Funktion)
- Wächst bei x86 von „oben“ nach „unten“, also zu niedrigeren Adressen hin

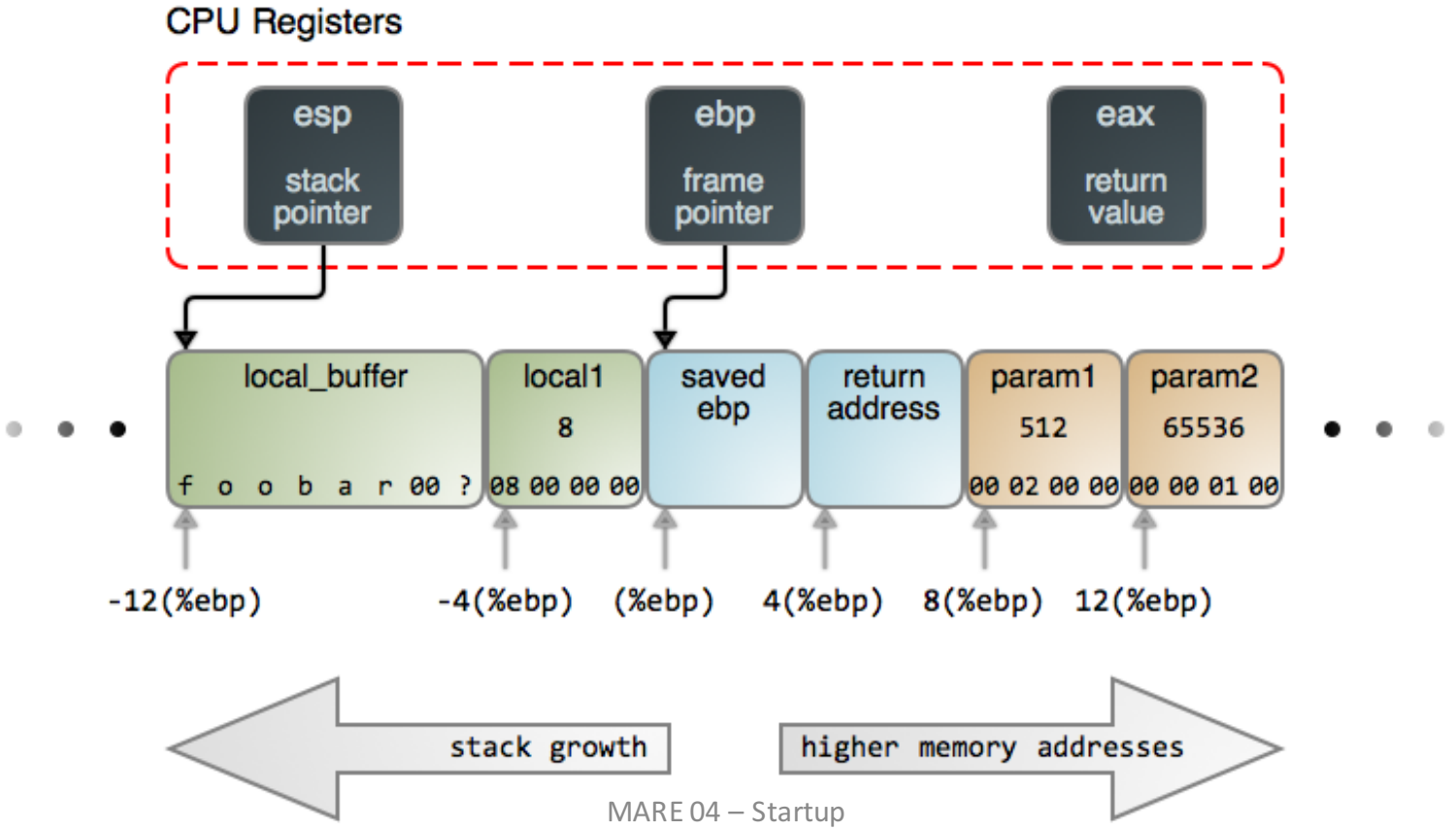
- Langfristig benötigte Informationen sind in globalen Datensektionen (.data, .rodata, .bss) und im Heap enthalten!

Stack-Aufbau (1)

http://duartes.org/gustavo/blog/post/journey-to-the-stack/

Jeder Funktionsaufruf erzeugt dynamisch neuen *stack frame*:

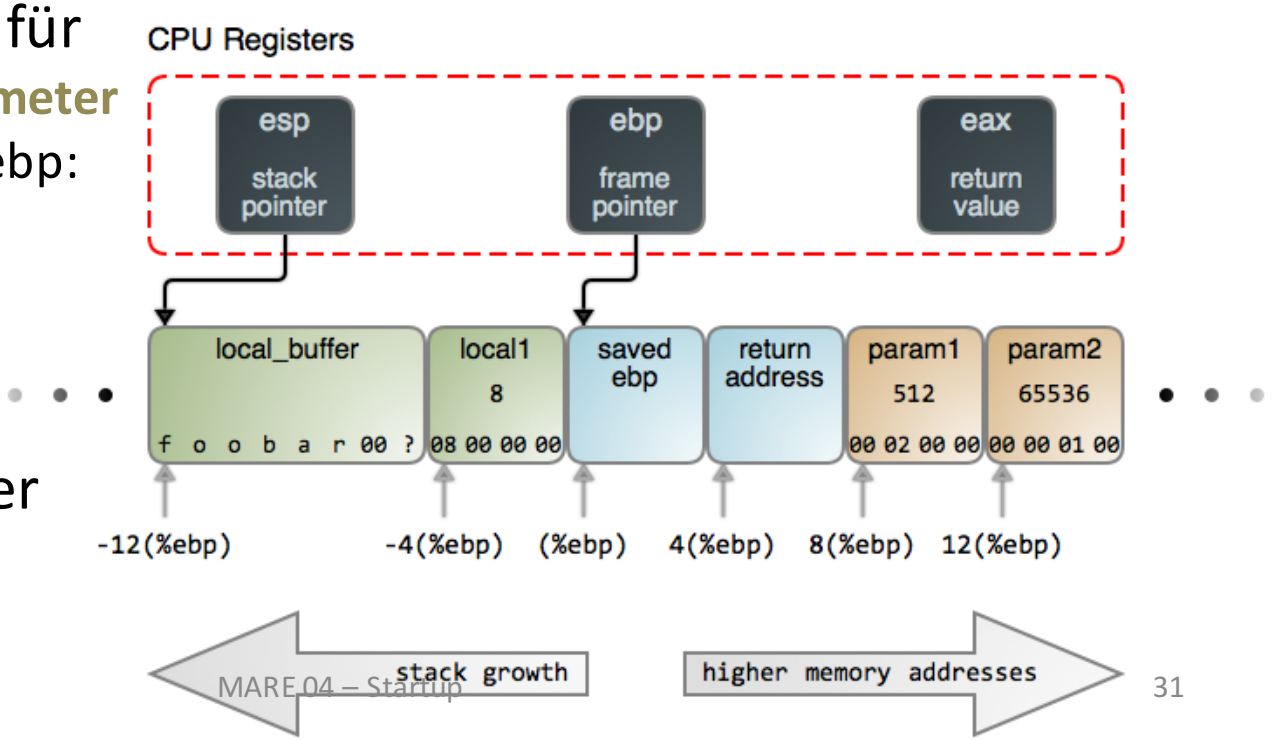
- Enthält Parameter, lokale Variablen, Kontextinformationen



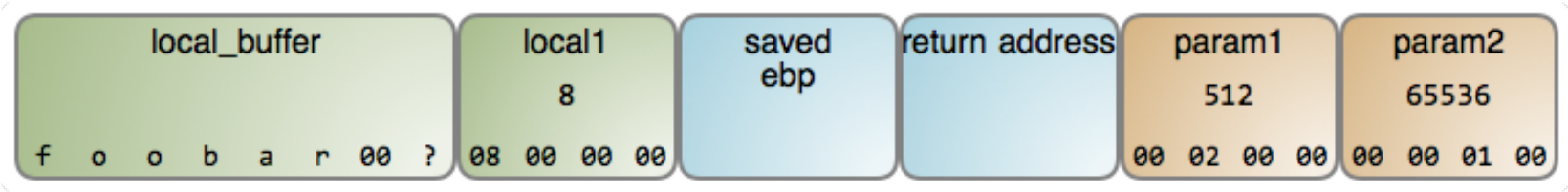
Stack-Aufbau (2)

Verwendete CPU-Register

- Stackpointer **esp**: Zeigt auf *letztes belegtes Element* im Stack
- Base- oder Framepointer **ebp**: Zeigt auf *stack frame* der aktuellen Funktion
- Referenzadresse für
  - **Funktionsparameter** (oberhalb von ebp: ab ebp+8)
  - **Lokale Variable** (unterhalb von ebp)
- Rückgabewert der Funktion in **eax**



Details des stack frame



Von links (niedrigere Adresse) nach rechts (höhere Adressen)

- Lokale Variablen:
  - local_buffer: 7 Byte langer String („foobar\0“) + Padding
  - local1: 4 Byte große Integer-Variable (Wert 8)
- Verwaltungsinformationen:
  - Gesicherter Framepointer (ebp) der aufrufenden Funktion
  - Rücksprungadresse zu aufrufender Funktion
- Funktionsparameter:
  - param1: 4 Byte-Integer, Wert 512
  - param2: 4 Byte-Integer, Wert 65536

```
int foo(int param1, int param2) {
   int local1;
   char local_buffer[7];
…
}
```

Erzeugen von stack frames (1)

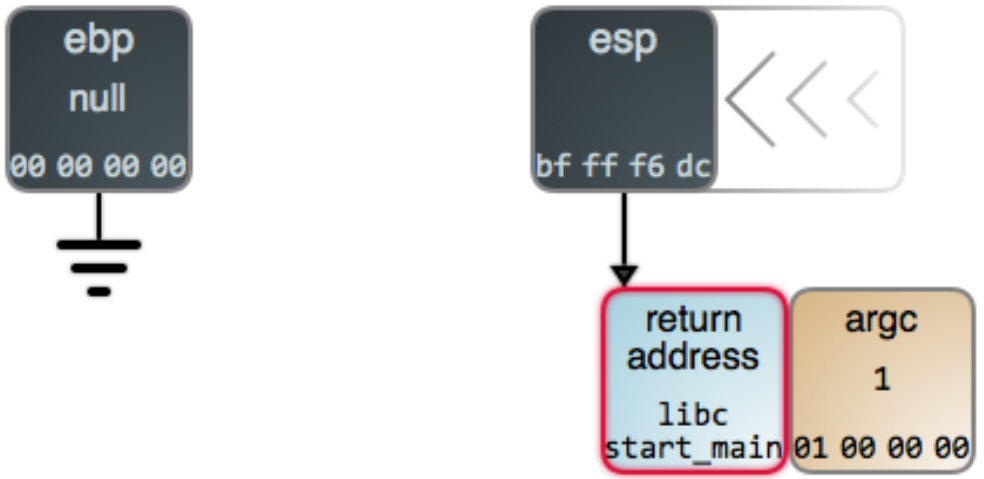
Compiler fügt Code zur Verwaltung von Stackframes ein:

- Prolog/Epilog: Verwalten von ebp und Raum für lokale Variablen

```
1.    call main # push return address onto stack, jump into main
```

```
int add(int a, int b)
{
  int result = a + b;
  return result;
}

int main(int argc)
{
  int answer;
  answer = add(40, 2);
}
```



Erzeugen von stack frames (2)

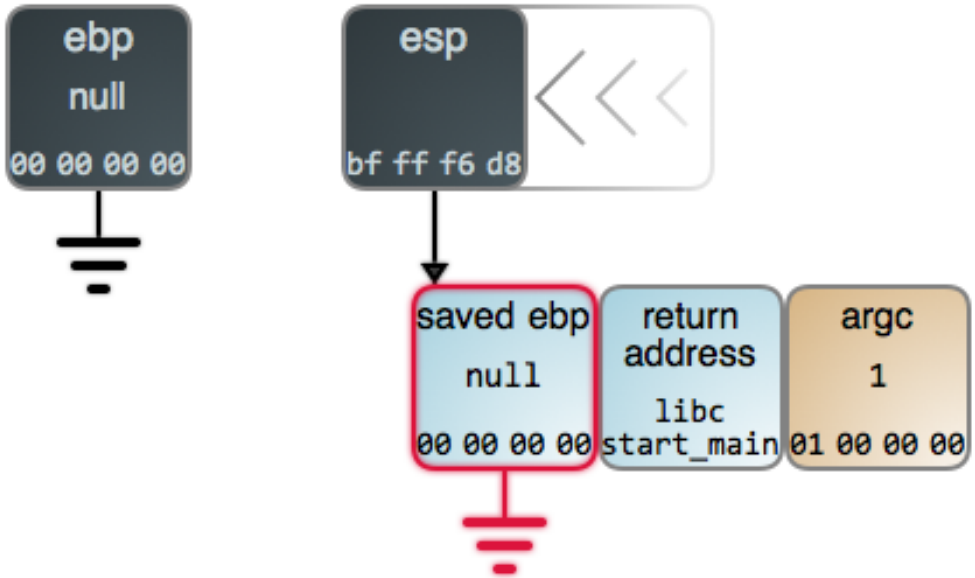
Framepointer ebp muss vor Aufruf gesichert werden

- Wird von aufgerufener Funktion überschrieben
- Hier: ebp=0 (main ist „oberste“ Funktion)

```
2.      pushl %ebp  # save current ebp register value
```

```
int add(int a, int b)
{
  int result = a + b;
  return result;
}

int main(int argc)
{
  int answer;
  answer = add(40, 2);
}
```



Erzeugen von stack frames (3)

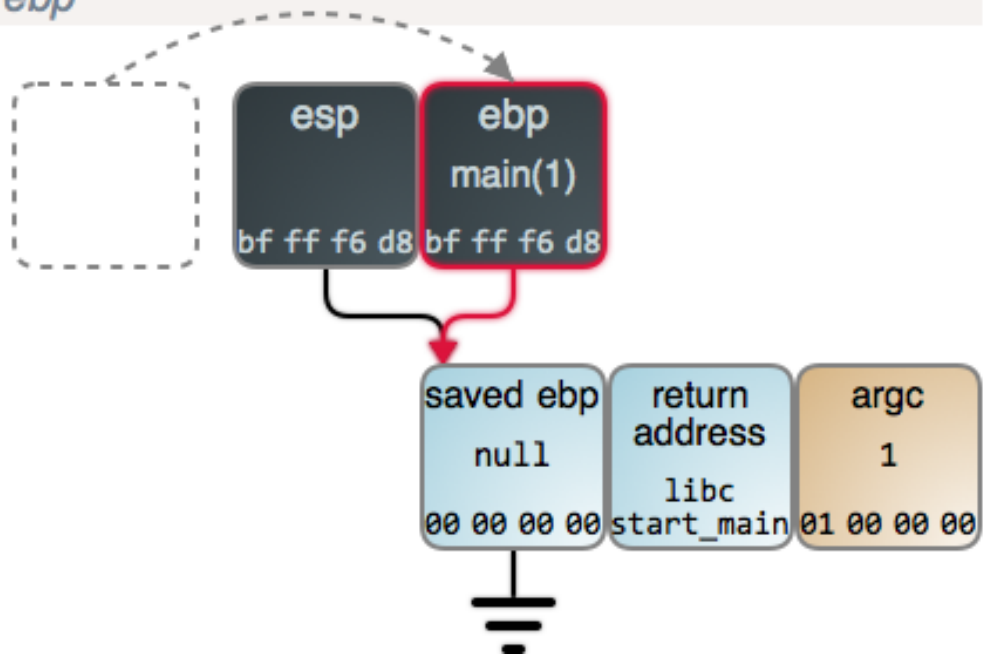
Setzen des neuen Framepointers

- Auf aktuelles „unteres“ Ende des Stacks

```
3.    movl %esp, %ebp # copy esp to ebp
```

```
int add(int a, int b)
{
  int result = a + b;
  return result;
}

int main(int argc)
{
  int answer;
  answer = add(40, 2);
}
```



Erzeugen von stack frames (4)

Schritte 2–4 sind der Funktionsprolog!

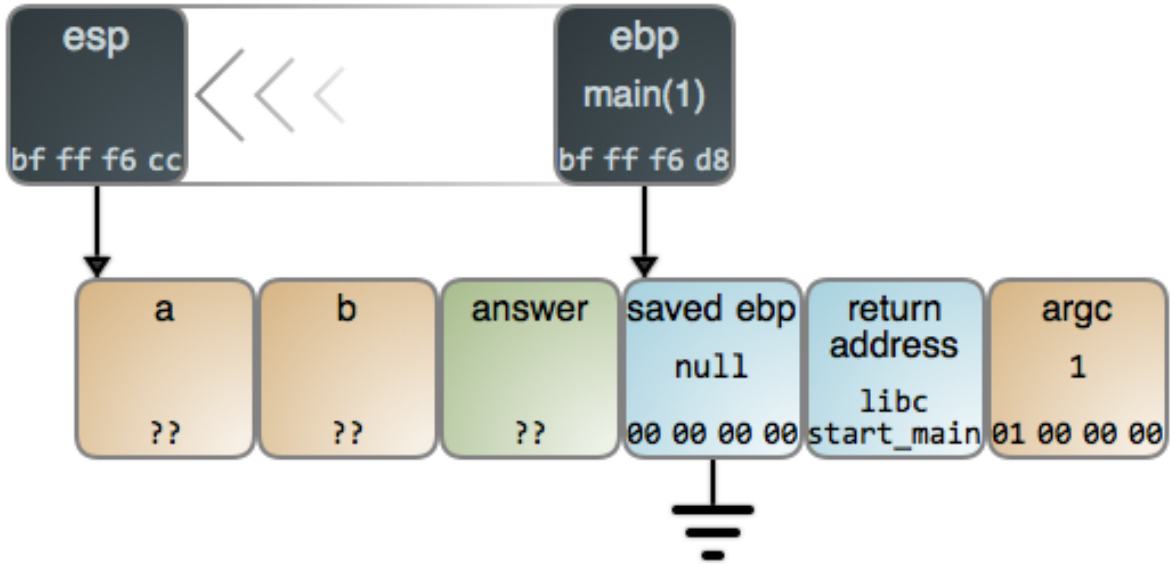
Stackpointer anpassen

- Platz auf Stack für lokale Variablen und Parameter schaffen

```
4.    subl $12, %esp # make room for stack data
```

```
int add(int a, int b)
{
  int result = a + b;
  return result;
}

int main(int argc)
{
  int answer;
  answer = add(40, 2);
}
```



Erzeugen von stack frames (5)

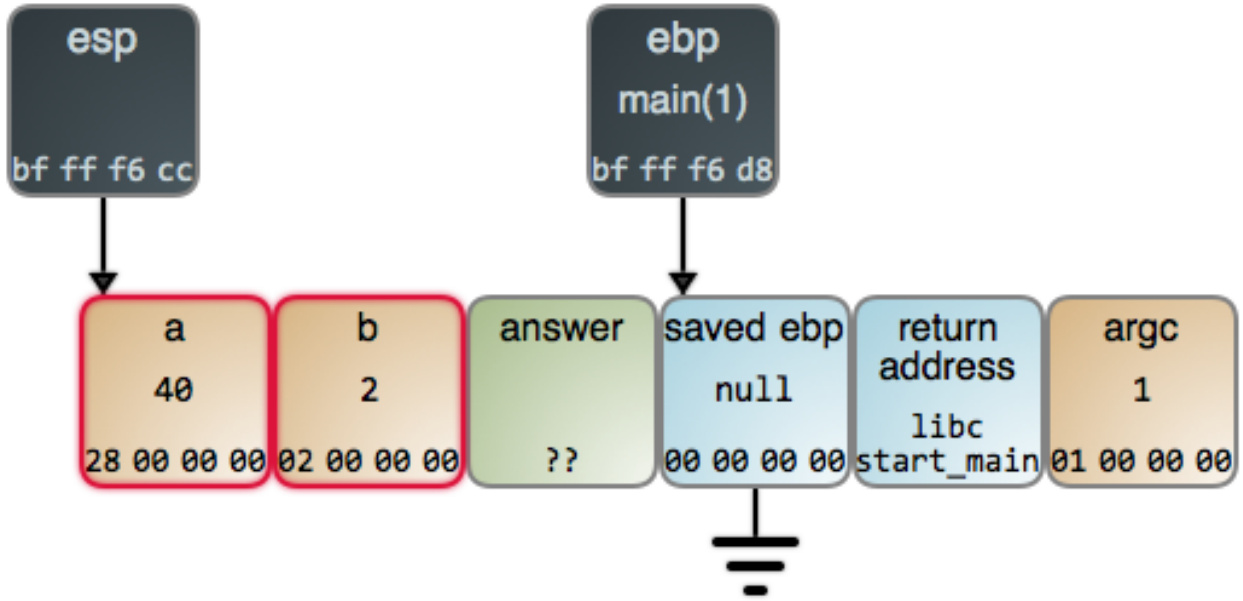
Parameter 40 und 2 für Funktionsaufruf von „add“ auf Stack

- Umgekehrte Reihenfolge, Adressierung relativ zu esp

```
5.      movl $2, 4(%esp) # set b to 2
        movl $40, (%esp) # set a to 40
```

```
int add(int a, int b)
{
  int result = a + b;
  return result;
}

int main(int argc)
{
  int answer;
  answer = add(40, 2);
}
```

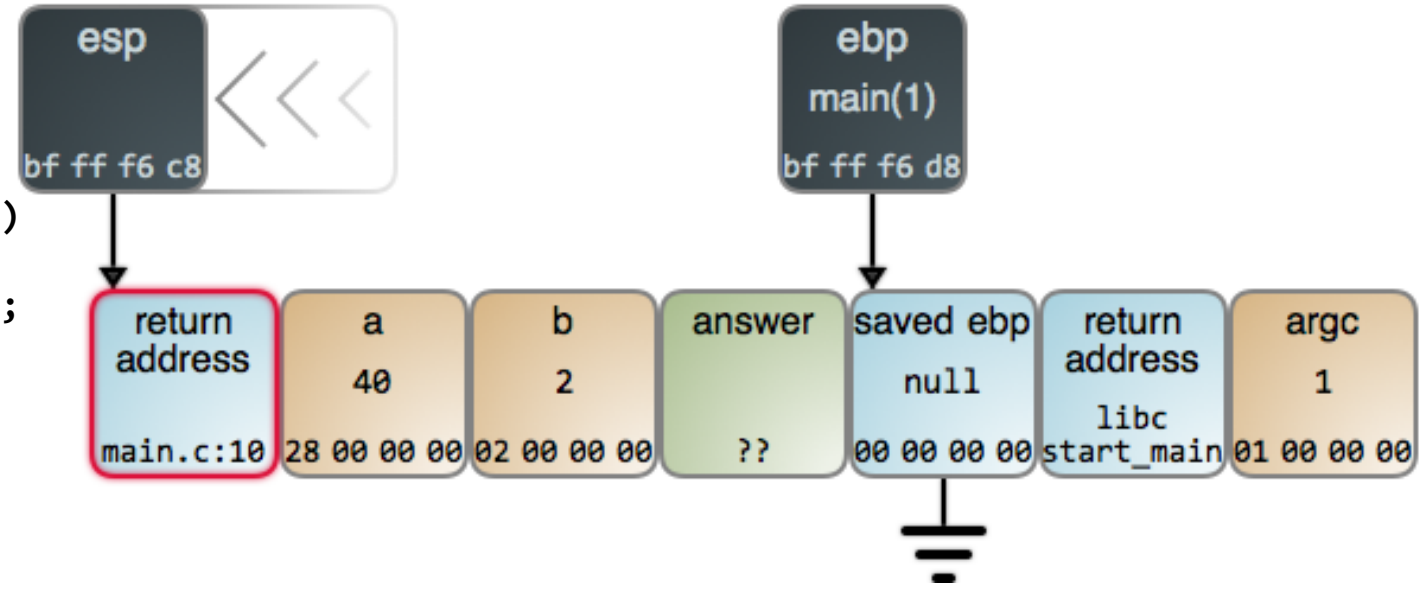


Erzeugen von stack frames (6)

Parameter 40 und 2 für Funktionsaufruf von „add“ auf Stack

- Umgekehrte Reihenfolge, Adressierung relativ zu esp

```
6.      call add # push return address onto stack, jump into add
```




```
int add(int a, int b)
{
  int result = a + b;
  return result;
}

int main(int argc)
{
  int answer;
  answer = add(40, 2);
}
```

Erzeugen von stack frames (7)

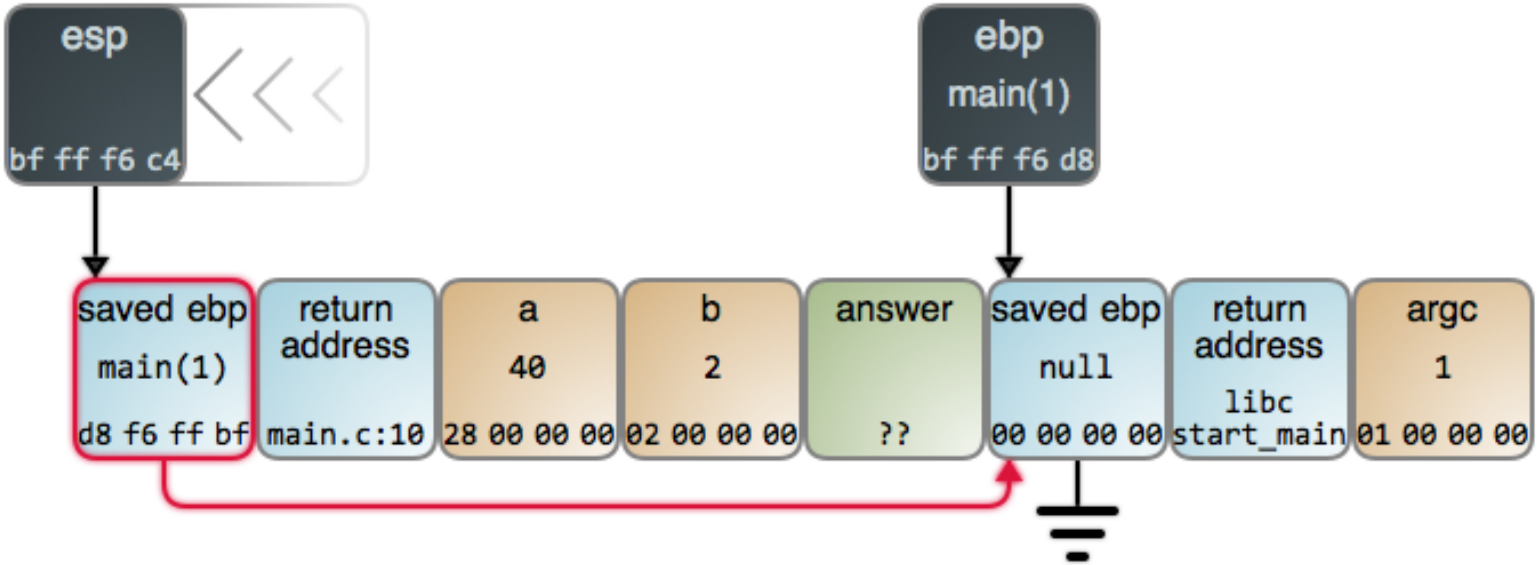
```
int add(int a, int b)
{
  int result = a + b;
  return result;
}

int main(int argc)
{
  int answer;
  answer = add(40, 2);
}
```

Neuer Prolog für „add“ geschaffen

- Stack = verkettete Liste von stack frames
- Platz für Sichern des vorherigen ebp

```
7.    pushl %ebp # save current ebp register value
```



Erzeugen von stack frames (8)

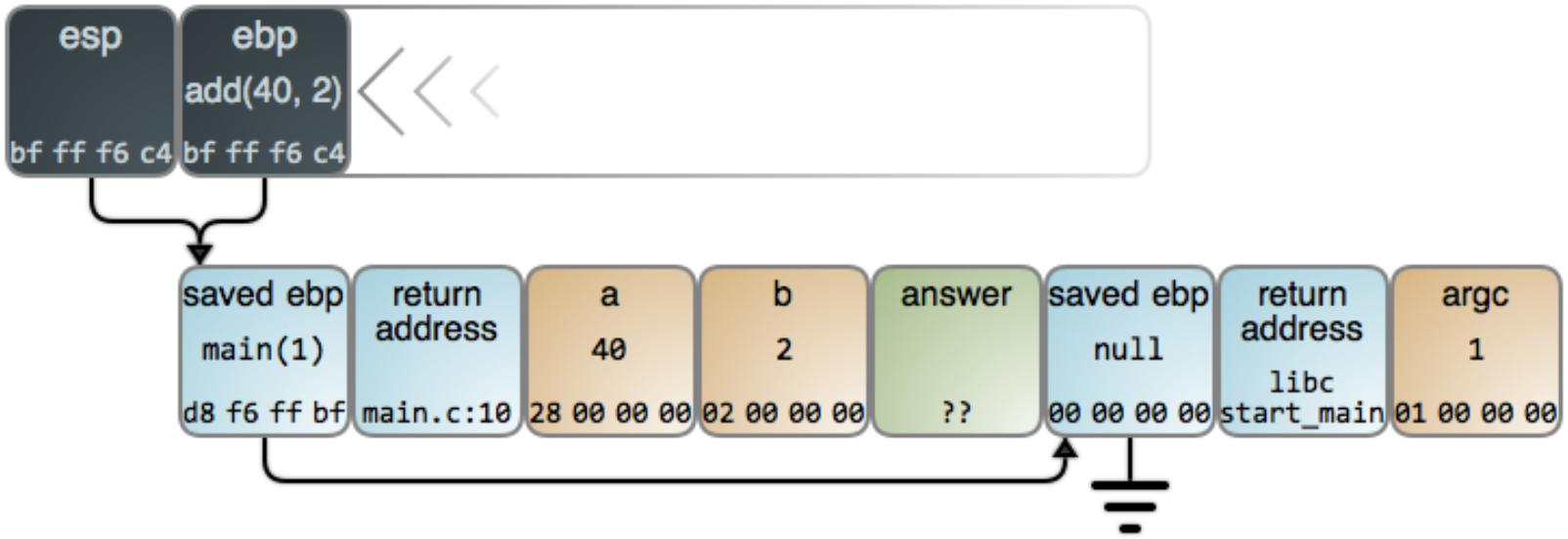
```
int add(int a, int b)
{
  int result = a + b;
  return result;
}

int main(int argc)
{
  int answer;
  answer = add(40, 2);
}
```

Alten Framepointer sichern

- Neuen Framepointer ebp auf Wert des aktuellen Stackpointers setzen

```
8.    movl %esp, %ebp # copy esp to ebp
```



Erzeugen von stack frames (9)

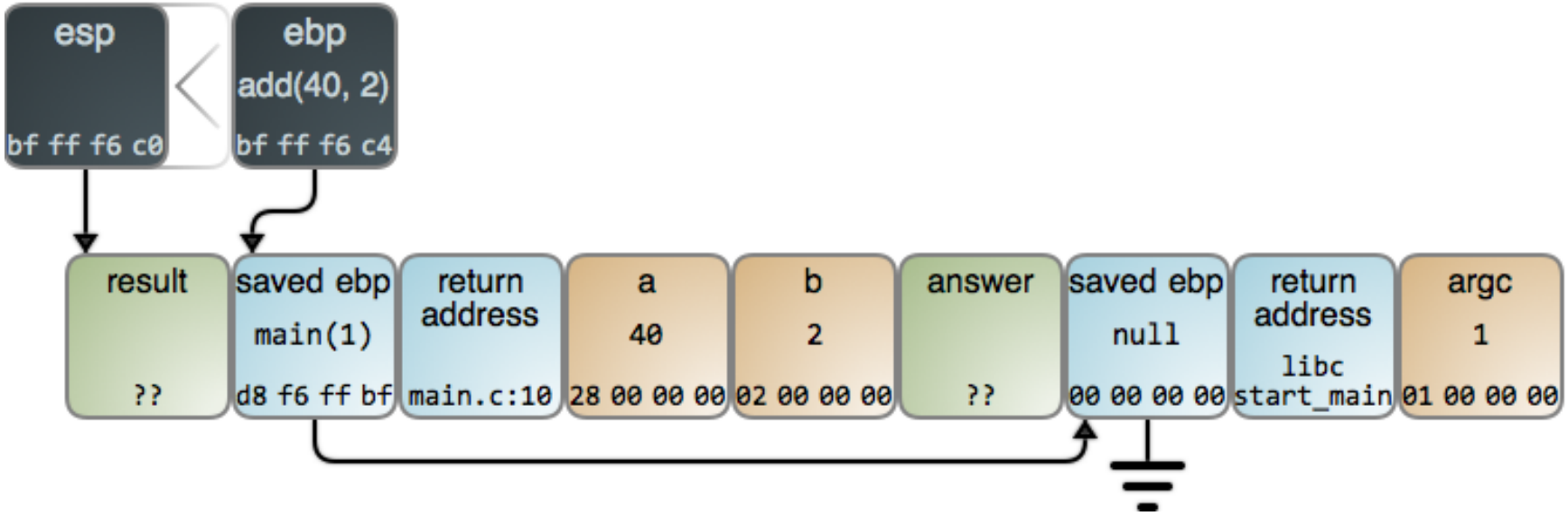
```
int add(int a, int b)
{
  int result = a + b;
  return result;
}

int main(int argc)
{
  int answer;
  answer = add(40, 2);
}
```

Platz für lokale Variable „result“ schaffen

- Stackpointer esp um 4 erniedrigen
- Framepointer ebp bleibt gleich!

```
9.    subl $4, %esp # make room for result
```



HOCHSCHULE COBURG

Erzeugen von stack frames (10)

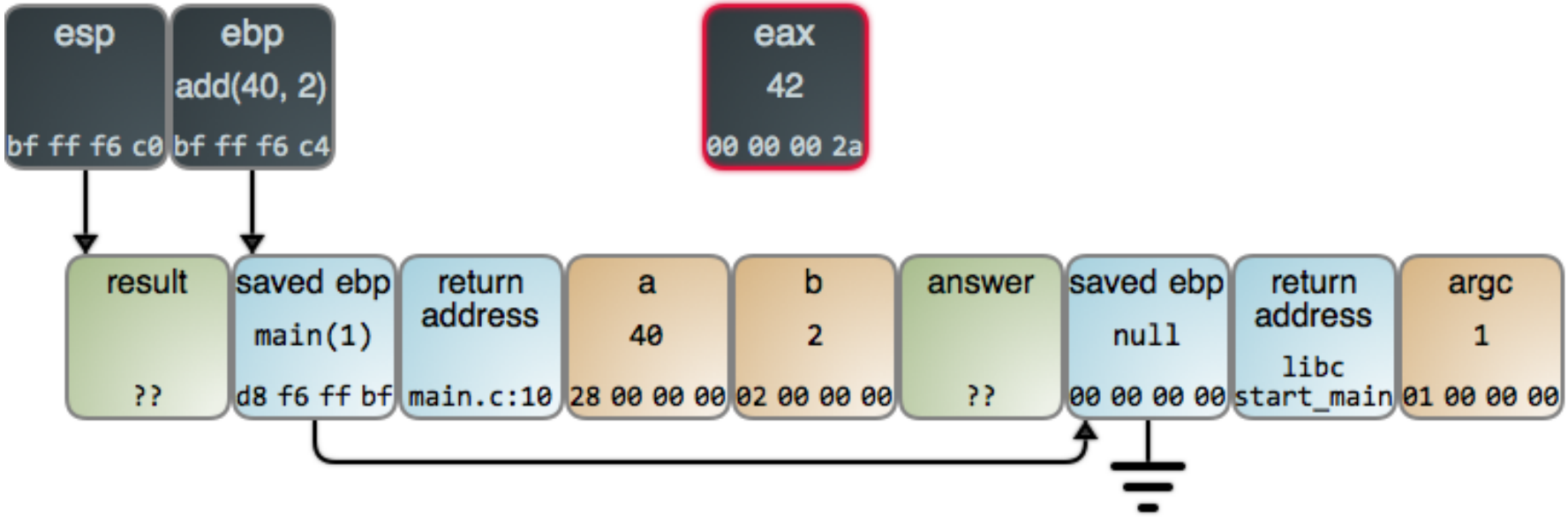
```
int add(int a, int b)
{
  int result = a + b;
  return result;
}

int main(int argc)
{
  int answer;
  answer = add(40, 2);
}
```

Parameter lesen und addieren

- Parameter in ebp+12 (b) und ebp+8 (a)

```
      movl 12(%ebp), %eax  # move b to eax
10.   movl 8(%ebp), %edx   # move a to edx
      addl %edx, %eax      # add edx into eax. total is 42.
```



Erzeugen von stack frames (11)

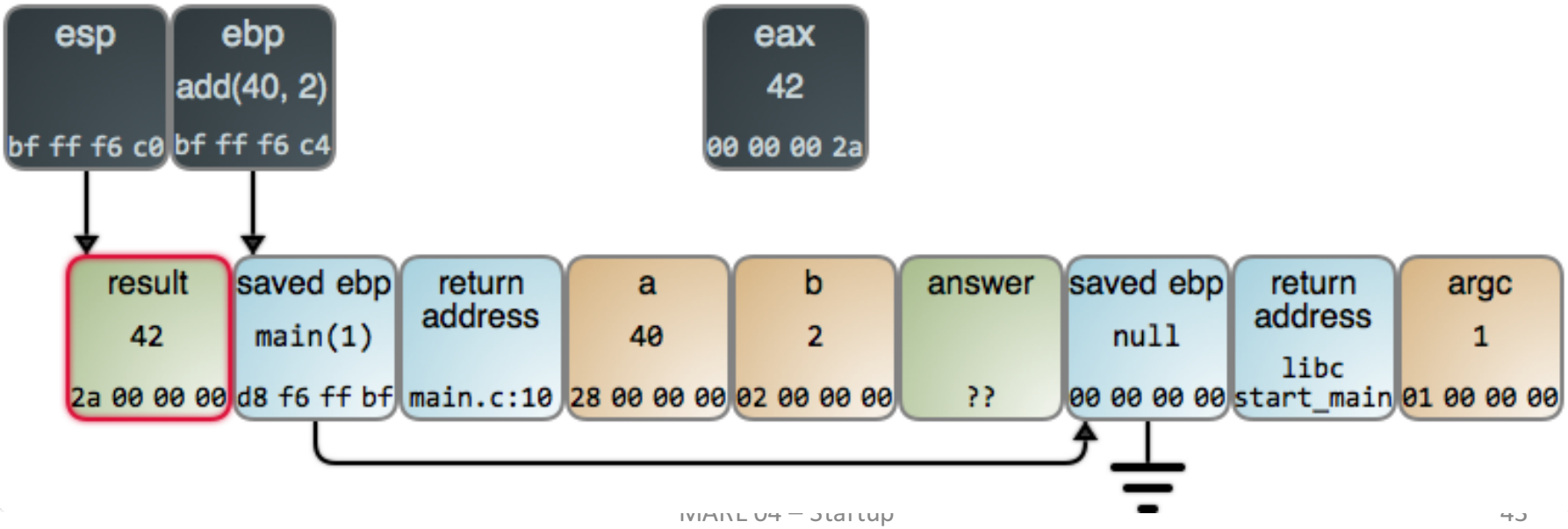
```
int add(int a, int b)
{
  int result = a + b;
  return result;
}

int main(int argc)
{
  int answer;
  answer = add(40, 2);
}
```

Ergebnis der Addition in lokale Variable „result“ schreiben

- Result liegt an Adresse ebp-4

```
11.    movl %eax, -4(%ebp) # copy eax to result
```



Fazit

Startup von Programmen ist aufwendiger Prozess

- Initialisierung, Konstruktoren, glibc...

Stack ist zentrales Konzept für hierarchische Funktionsaufrufe

- Funktionsparameter
- Rücksprungadressen
- Lokale Variable
- Aufrufhierarchie

Literatur

Unix-Prozessmodell, Prozess-Startup

- Eric S. Raymond, *„The Art of UNIX Programming“,* Addison-Wesley 2003, ISBN-13: 978-0131429017

x86-Assembler (32/64 Bit)

- http://www.cs.umd.edu/~meesh/cmsc311/links/handouts/ia32.pdf
- https://www.cs.cmu.edu/~fp/courses/15213-s07/misc/asm64-handout.pdf

Stack

- Giuseppe Di Cataldo, *„Stack Frames – A Look From Inside“,* Apress 2016, ISBN-13: 978-1-4842-2181-5

ABI-Konventionen

- https://sites.google.com/site/x32abi/

Anhang: Wo ist envp?

Bei Aufruf von _libc_start_main wird envp nicht übergeben!

- Prototyp:
  `int main(int argc, char** argv, char** envp);`
- envp = Zeiger auf Array von Strings mit Environmentvariablen

- Dieses Array fängt direkt hinter den Strings von argv an:
  - _libc_init_first wird von _libc_start_main aufgerufen

```
void __libc_init_first(int argc, char *arg0, ...)
{
    char **argv = &arg0, **envp = &argv[argc + 1];
    __environ = envp;
    __libc_init (argc, argv, envp);
}
```

Anhang: Was kommt *nach* envp?

Weitere Informationen aus ELF-Header ebenfalls übergeben:

- Array fängt direkt ***hinter den Strings von envp*** an:

```
$ LD_SHOW_AUXV=1 ./prog1
AT_SYSINFO:      0xe62414
AT_SYSINFO_EHDR: 0xe62000
AT_HWCAP:    fpu vme de pse tsc msr pae mce cx8 apic
             mtrr pge mca cmov pat pse36 clflush dts
             acpi mmx fxsr sse sse2 ss ht tm pbe
AT_PAGESZ:       4096
AT_CLKTCK:       100
AT_PHDR:         0x8048034
AT_PHENT:        32
AT_PHNUM:        8
AT_BASE:         0x686000
AT_FLAGS:        0x0
AT_ENTRY:        0x80482e0
AT_UID:          1002
AT_EUID:         1002
AT_GID:          1000
AT_EGID:         1000
AT_SECURE:       0
AT_RANDOM:       0xbff09acb
AT_EXECFN:       ./prog1
AT_PLATFORM:     i686
```