NTNU | Norwegian University of Science and Technology

Operating Systems

Lecture 8:
From Source Code to Process

Michael Engel

From Source Code to Process

- The compilation process
  - Compiler, assembler, linker
  - The ELF file format
- Loading a program
  - Dynamic loader
  - Shared libraries
- Running a program
  - What comes before main?

Compiling code

Preprocessor

- Expands `#include`s and macros

Compiler

- Generates assembler source code from C source code

Assembler

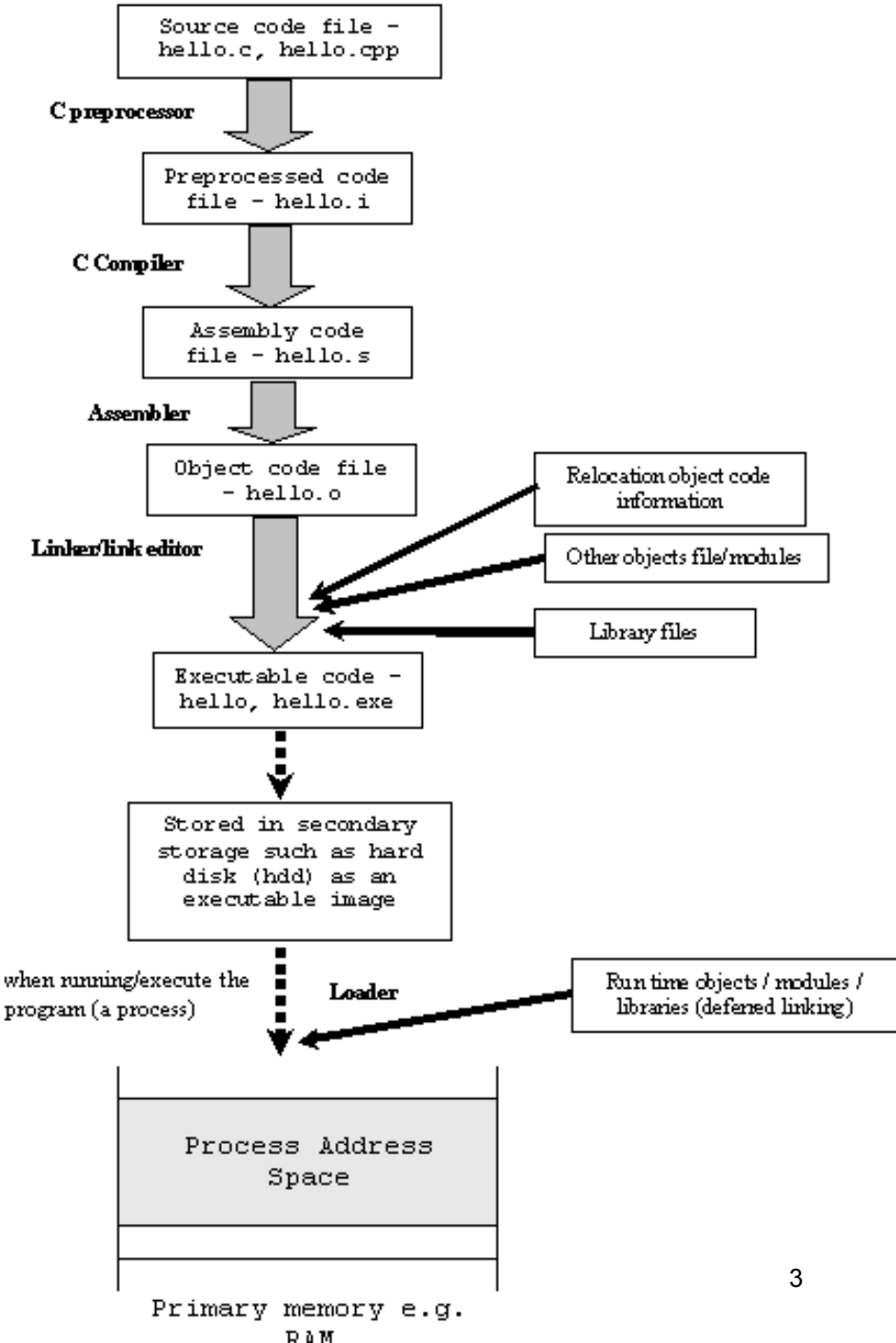
- Generates *object code* from assembler source code

Linker

- Combines (one or) multiple object files (+ libraries) to an executable file

Loader

- Loads executable file into main memory



Example: From .c to .o

```c
#include <stdio.h>
static void display(int i, int *ptr);

int main(void)
{
      int x = 5;
      int *xptr = &x;
      printf("In main() program:\n");
      printf("x value is %d and is stored at address %p.\n", x, &x);
      printf("xptr pointer points to address %p which holds a value of %d.\n",
           xptr, *xptr);
      display(x, xptr);
      return 0;
}

void display(int y, int *yptr)
{
      char var[7] = "ABCDEF";
      printf("In display() function:\n");
      printf("y value is %d and is stored at address %p.\n", y, &y);
      printf("yptr pointer points to address %p which holds a value of %d.\n",
           yptr, *yptr);
}
```

Example: From .c to .o (2)

```
$ gcc -c testprog1.c
```

compiler: translates C source code file to object file `testprog.o`

```
$ ls -l
total 8
-rw-r--r-- 1 me me  629 Mar 22 13:15 testprog1.c
-rw-r--r-- 1 me me 1412 Mar 22 13:16 testprog1.o
```

check the type of the object file with `file`

```
$ file testprog1.o
testprog1.o: ELF 32-bit LSB relocatable, intel 80386, version 1 (SYSV), not
stripped
```

look at the contents of the object file with `hexdump`

```
$ hexdump -C testprog1.o
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  01 00 03 00 01 00 00 00  00 00 00 00 00 00 00 00  |................|
00000020  84 02 00 00 00 00 00 00  34 00 00 00 00 00 28 00  |........4.....(.|
00000030  0b 00 08 00 8d 4c 24 04  83 e4 f0 ff 71 fc 55 89  |.....L$.....q.U.|
[…]
```

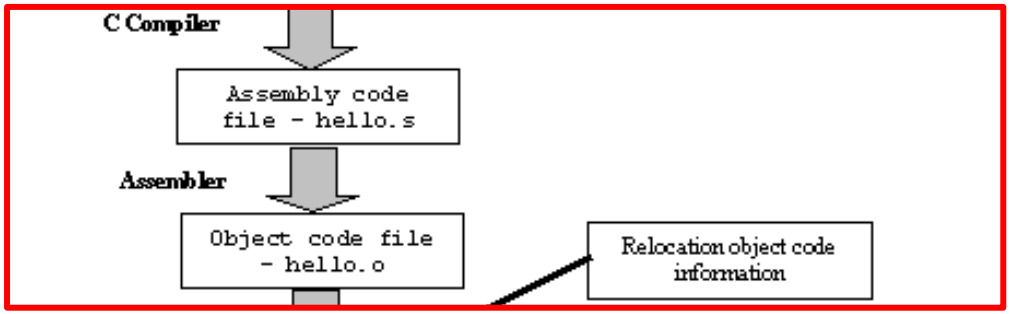
hexadecimal byte values

printable ASCII char's

Wait a moment…

Where is our assembler source?

Only generated internally as temporary file!



Compiler option "`-S`" explicitly generates a `.s` file:

```
$ gcc -S testprog1.c
$ ls -1
testprog1.c
testprog1.o
testprog1.s
```

```
        .file   "testprog1.c"
        .section        .rodata
.LC0:
        .string "In main() program:"
        .align 4
.LC1:
        .string "x value is %d and is stored at address %p.\n"
        .align 4
.LC2:
        .string "xptr pointer points to address %p which holds a value of %d.\n"
        .text
.globl main
        .type   main, @function
main:
        leal    4(%esp), %ecx
        andl    $-16, %esp
        pushl   -4(%ecx)
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ecx
        subl    $36, %esp
        movl    $5, -12(%ebp)
        leal    -12(%ebp), %eax
```

ELF internals (1)

```
$ file testprog1.o
testprog1.o: ELF 32-bit LSB relocatable, intel 80386, version 1 (SYSV),
not stripped
```

ELF:
Executable and Linking Format

```
$ man elf
[...]
The ELF header is described by the type Elf32_Ehdr or Elf64_Ehdr:
#define EI_NIDENT 16
typedef struct {
    unsigned char e_ident[EI_NIDENT];
    uint16_t      e_type;
    uint16_t      e_machine;
    uint32_t      e_version;
    ElfN_Addr     e_entry;
    ElfN_Off      e_phoff;
    ElfN_Off      e_shoff;
    uint32_t      e_flags;
    uint16_t      e_ehsize;
    uint16_t      e_phentsize;
    uint16_t      e_phnum;
    uint16_t      e_shentsize;
    uint16_t      e_shnum;
    uint16_t      e_shstrndx;
} ElfN_Ehdr;
```

ELF internals (2)

```
$ hexdump -C testprog1.o
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  01 00 03 00 01 00 00 00  00 00 00 00 00 00 00 00  |................|
00000020  84 02 00 00 00 00 00 00  34 00 00 00 00 00 28 00  |........4.....(.|
00000030  0b 00 08 00 8d 4c 24 04  83 e4 f0 ff 71 fc 55 89  |.....L$.....q.U.|
[…]
```

```
00000000  7f 45 4c 46 ; ELF „Magic“
00000004  01
00000005  01
00000006  01
00000007  00 00 00 00 00 00 00 00 00
```

/usr/include/elf.h:

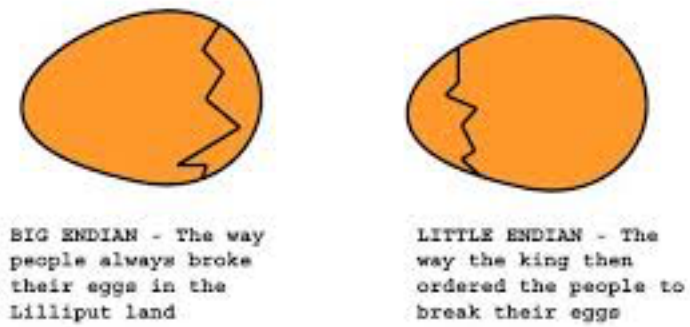
```
#define ELFMAG        "\177ELF" // \177: octal
#define EI_CLASS      4 /* File class byte index
#define ELFCLASSNONE  0 /* Invalid class */
#define ELFCLASS32    1 /* 32-bit objects */
#define ELFCLASS64    2 /* 64-bit objects */
#define ELFCLASSNUM   3
```

```
#define EI_NIDENT 16
typedef struct {
    unsigned char e_ident[EI_NIDENT];
    uint16_t      e_type;
    uint16_t      e_machine;
    uint32_t      e_version;
    ElfN_Addr     e_entry;
    ElfN_Off      e_phoff;
    ElfN_Off      e_shoff;
    uint32_t      e_flags;
    uint16_t      e_ehsize;
    uint16_t      e_phentsize;
    uint16_t      e_phnum;
    uint16_t      e_shentsize;
    uint16_t      e_shnum;
    uint16_t      e_shstrndx;
} ElfN_Ehdr;
```

Endianness

/usr/include/elf.h:

```
#define EI_DATA          5    /* Data encoding byte index */
#define ELFDATANONE      0    /* Invalid data encoding */
#define ELFDATA2LSB      1    /* 2's complement, little endian */
#define ELFDATA2MSB      2    /* 2's complement, big endian */
```



Endianness or byte order

- Order in which the bytes of a data type using more than 8 bit are stored in memory
- Example:

  Decimal number 123456789 = hexadecimal **0x075BCD15**

Little endian: least significant byte first
(at lowest address)

00000000 **15**
00000001 **CD**
00000002 **5B**
00000003 **07**

⇔

Big Endian: most significant byte first:

00000000 **07**
00000001 **5B**
00000002 **CD**
00000003 **15**

Dissect an ELF with readelf

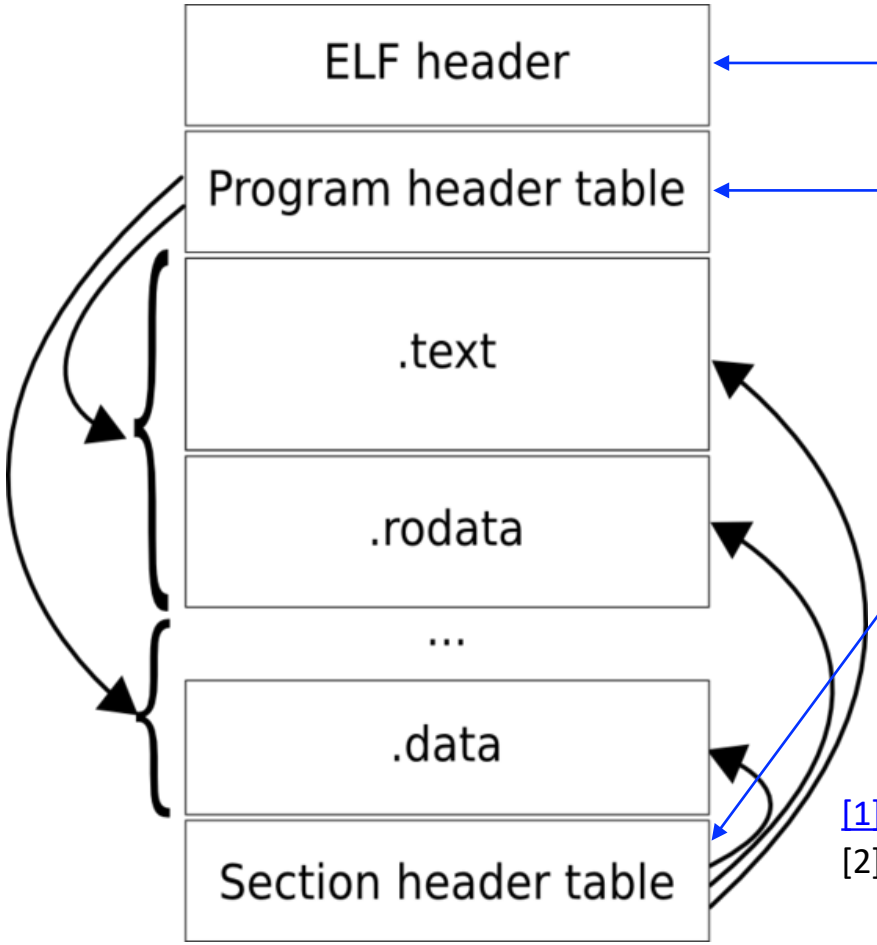
```
$ readelf testprog1.o
Usage: readelf <option(s)> elf-file(s)
 Display information about the contents of ELF format files
 Options are:
  -a --all               Equivalent to: -h -l -S -s -r -d -V -A -I
  -h --file-header       Display the ELF file header
[...]
```

```
$ readelf –h testprog1.o
```

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          644 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           40 (bytes)
  Number of section headers:         11
  Section header string table index: 8
```

ELF file structure

ELF file format is standardized (see [1] and [2])



Header with general information (see previous slides)

Information on the ELF file segments

Table of section headers with detailed information on each segment

[1] *System V Application Binary Interface* Edition 4.1 (1997-03-18)

[2] Tool Interface Standard (TIS)
Executable and Linking Format (ELF) Specification Version 1.2 (May 1995)

ELF sections

```
$ readelf -S testprog1.o
There are 11 section headers, starting at offset 0x284:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 0000db 00  AX  0   0  4
  [ 2] .rel.text         REL             00000000 000524 000060 08      9   1  4
  [ 3] .data             PROGBITS        00000000 000110 000000 00  WA  0   0  4
  [ 4] .bss              NOBITS          00000000 000110 000000 00  WA  0   0  4
  [ 5] .rodata           PROGBITS        00000000 000110 000102 00   A  0   0  4
  [ 6] .comment          PROGBITS        00000000 000212 00001f 00      0   0  1
  [ 7] .note.GNU-stack   PROGBITS        00000000 000231 000000 00      0   0  1
  [ 8] .shstrtab         STRTAB          00000000 000231 000051 00      0   0  1
  [ 9] .symtab           SYMTAB          00000000 00043c 0000c0 10     10   9  4
  [10] .strtab           STRTAB          00000000 0004fc 000026 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

ELF sections: what goes where?

What is where in the object file?

$ **gcc –c foo.c**

$ **readelf –S foo.o**

```
There are 11 section headers, starting at offset 0xd8:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 00002c 00  AX  0   0  4
  [ 2] .rel.text         REL             00000000 000364 000020 08      9   1  4
  [ 3] .data             PROGBITS        00000000 000060 000004 00  WA  0   0  4
  [ 4] .bss              NOBITS          00000000 000064 000004 00  WA  0   0  4
  [ 5] .rodata           PROGBITS        00000000 000064 000004 00   A  0   0  4
```

Example program foo.c:

```c
const int a = 42;
int b = 23;
int c;

int main(void) {
  c = a + b;
  return c;
}
```

Section	Function
text (.text)	Machine code (instructions) and entry point (address)
read only data (.rodata)	initialized constants
read/write data (.data)	initialized variables
Base Storage Segment (.bss)	uninitialised data
Symbols (.symtab)	addresses for symbolic names

Both .data and .rodata hold a variable using 4 bytes each = sizeof(int)

ELF section details

Offsets in ELF .o file:
no memory addresses!

```
There are 11 section headers, starting at offset 0xd8:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 00002c 00  AX  0   0  4
  [ 2] .rel.text         REL             00000000 000364 000020 08      9   1  4
  [ 3] .data             PROGBITS        00000000 000060 000004 00  WA  0   0  4
  [ 4] .bss              NOBITS          00000000 000064 000000 00  WA  0   0  4
  [ 5] .rodata           PROGBITS        00000000 000064 000004 00   A  0   0  4

00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  01 00 03 00 01 00 00 00  00 00 00 00 00 00 00 00  |................|
00000020  d8 00 00 00 00 00 00 00  34 00 00 00 00 00 28 00  |........4.....(.|
00000030  0b 00 08 00 8d 4c 24 04  83 e4 f0 ff 71 fc 55 89  |.....L$.....q.U.|
00000040  e5 51 8b 15 00 00 00 00  a1 00 00 00 00 8d 04 02  |.Q..............|
00000050  a3 00 00 00 00 a1 00 00  00 00 59 5d 8d 61 fc c3  |..........Y].a..|
00000060  17 00 00 00 2a 00 00 00  00 47 43 43 3a 20 28 44  |....*....GCC: (D|
00000070  65 62 69 61 6e 20 34 2e  33 2e 32 2d 31 2e 31 29  |ebian 4.3.2-1.1)|
```

At address:

0x60: 17 00 00 00 => 0x00000017 = 23_{10}

0x64: 2a 00 00 00 => 0x0000002a = 42_{10}

ELF sections and assembler code

In the assembler code:

$ **gcc –S foo.c**

$ **cat foo.s**

```
    .file    "foo.c"
.globl a
    .section .rodata
    .align 4
    .type    a, @object
    .size    a, 4
a:
    .long    42
.globl b
    .data
    .align 4
    .type    b, @object
    .size    b, 4
b:
    .long    23
```

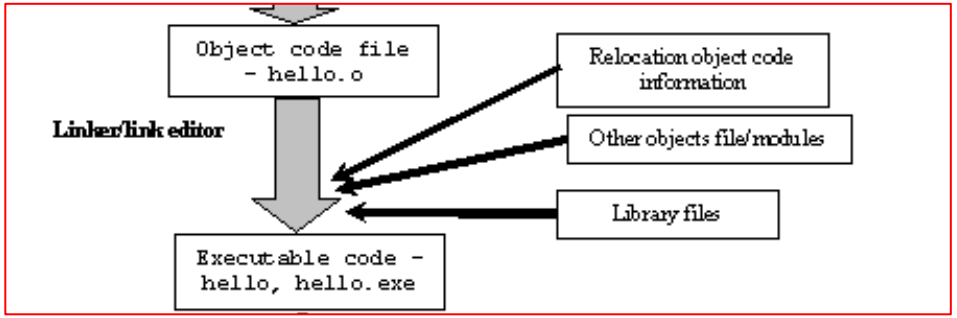
```
    .text
.globl main
    .type    main, @function
main:
    leal     4(%esp), %ecx
    andl     $-16, %esp
    pushl    -4(%ecx)
    pushl    %ebp
    movl     %esp, %ebp
    pushl    %ecx
    movl     a, %edx
    movl     b, %eax
    leal     (%edx,%eax), %eax
    movl     %eax, c
    movl     c, %eax
    popl     %ecx
    popl     %ebp
    leal     -4(%ecx), %esp
    ret
    .size    main, .-main
    .comm    c,4,4
    .ident   "GCC: (Debian 4.3.2-1.1) 4.3.2"
    .section .note.GNU-stack,"",@progbits
```

Example program foo.c:

```
const int a = 42;
int b = 23;
int c;

int main(void) {
  c = a + b;
  return c;
}
```

Object files – and then?

.o object files cannot be executed directly!



- Important parts of an executable file are missing
  - crt0 – startup code
  - initialization
    - variables in .bss are initialized (to 0)
    - for C++: calling of constructors
  - jump to "main" function and parameter passing (argc, argv, envp)
  - libraries, e.g. libc (C standard library), have to be added

- **Addresses of variables and functions are not resolved**
  - One of the main tasks of the linker

Symbols and addresses in object files

Section	Function
Symbols (.symtab)	Addresses for symbolic names

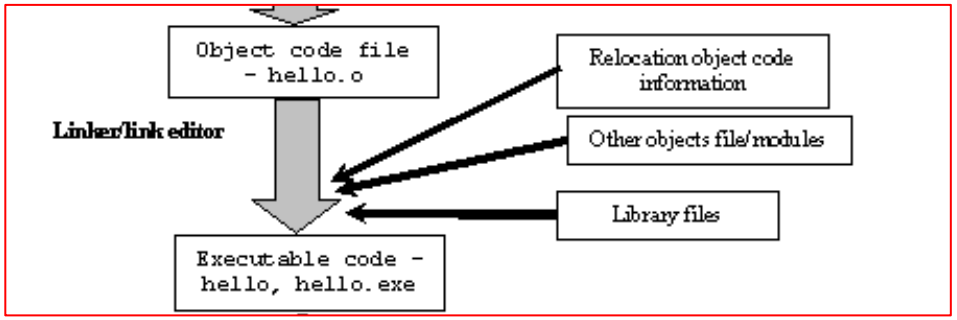
Addresses of functions and (global) variables have to be known

- Symbol table: assigns addresses to symbolic names for functions and variables:

```
$ readelf -s foo.o

Symbol table '.symtab' contains 12 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS foo.c
     2: 00000000     0 SECTION LOCAL  DEFAULT    1
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    4
     5: 00000000     0 SECTION LOCAL  DEFAULT    5
     6: 00000000     0 SECTION LOCAL  DEFAULT    7
     7: 00000000     0 SECTION LOCAL  DEFAULT    6
     8: 00000000     4 OBJECT  GLOBAL DEFAULT    5 a
     9: 00000000     4 OBJECT  GLOBAL DEFAULT    3 b
    10: 00000000    44 FUNC    GLOBAL DEFAULT    1 main
    11: 00000004     4 OBJECT  GLOBAL DEFAULT  COM c
```

- Addresses are set to 0 in the object file

Linker and relocations



Linker is passed a list of all object files required to build a program

- .o files (generated by the compiler or assembler): *object files*
  - can also be generated by C++, Fortran, ... compilers!
- .a files: *"archives"* of object files
  - *static libraries*
- .so files: *"shared object"s*
  - d*ynamic libraries* (on Windows: DLL "Dynamic Loadable Library")

Linker assigns text and data segments of the single .o files to parts of the address space for loading by the OS

- It resolves *references* to symbols (variables, functions)
- Controlled by a *linker script* (configuration file)

Linker and symbol resolution

Section	Function
Symbols (.symtab)	Addresses for symbolic names

Symbol table of the linked program

- Also contains symbols of all linked libraries – a bit confusing…

"gcc" is not the compiler itself, but only a front end which calls the preprozessor, compiler, linker… as required. Here, we link foo.o (+libraries) to the executable program foo!

```
$ gcc –o foo foo.o

$ readelf -s foo

Symbol table '.symtab' contains 76 entries:
Num:    Value  Size Type    Bind   Vis      Ndx Name
[…]
 64: 08048460     4 OBJECT  GLOBAL DEFAULT   15 a
[…]
 69: 0804956c     4 OBJECT  GLOBAL DEFAULT   23 b
[…]
 73: 08048374    44 FUNC    GLOBAL DEFAULT   13 main
 74: 08048254     0 FUNC    GLOBAL DEFAULT   11 _init
 75: 08049578     4 OBJECT  GLOBAL DEFAULT   24 c
```

Linker and symbol resolution (2)

Section	Function
text (.text)	Machine code (instructions) and entry point (address)
read only data (.rodata)	initialized constants
read/write data (.data)	initialized variables
Base Storage Segment (.bss)	uninitialised data
Symbols (.symtab)	addresses for symbolic names

Symbol table of the linked program

- Alternative: use "nm" ("names") from GNU binutils

```
$ gcc –o foo foo.o
$ nm foo
[…]
08048254 T _init
080482c0 T _start
08048460 R a
0804956c D b
08049578 B c
[…]
08048374 T main
```

Code	Section
T	.text
R	.rodata
D	.data
B	.bss

…compare:

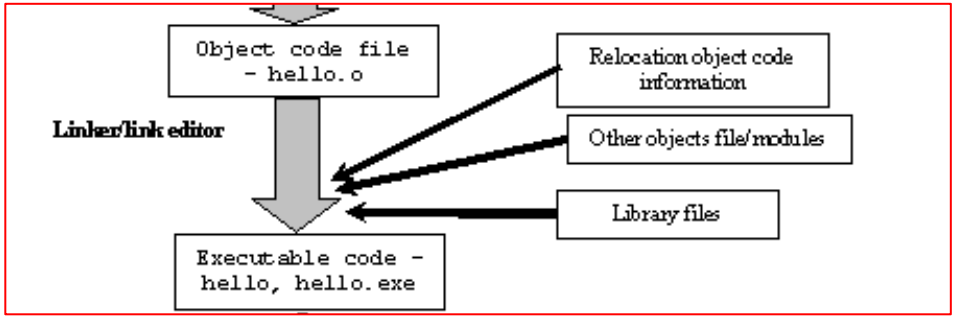
```
$ readelf -s foo
Symbol table '.symtab' contains 76 entries:
Num:    Value  Size Type    Bind   Vis      Ndx Name
[…]
 64: 08048460     4 OBJECT  GLOBAL DEFAULT   15 a
 69: 0804956c     4 OBJECT  GLOBAL DEFAULT   23 b
 73: 08048374    44 FUNC    GLOBAL DEFAULT   13 main
 74: 08048254     0 FUNC    GLOBAL DEFAULT   11 _init
 75: 08049578     4 OBJECT  GLOBAL DEFAULT   24 c
```

Linker and symbol resolution: .o files

```
$ objdump -d foo.o
```

foo.o – object file

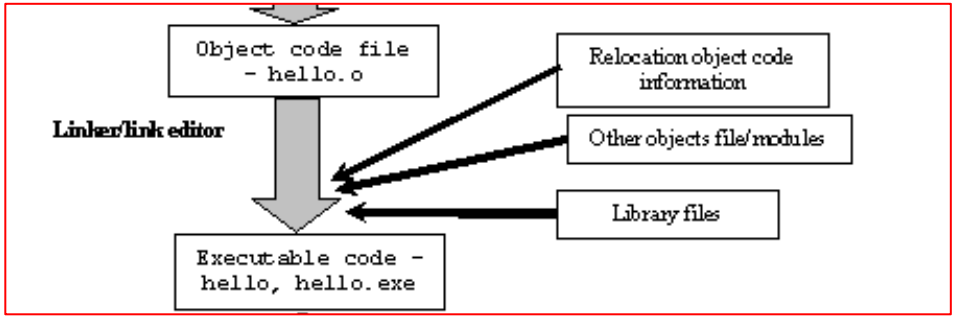
```
foo.o:     file format elf32-i386

Disassembly of section .text:

00000000 <main>:
   0:   8d 4c 24 04             lea    0x4(%esp),%ecx
   4:   83 e4 f0                and    $0xfffffff0,%esp
   7:   ff 71 fc                pushl  -0x4(%ecx)
   a:   55                      push   %ebp
   b:   89 e5                   mov    %esp,%ebp
   d:   51                      push   %ecx
   e:   8b 15 00 00 00 00       mov    0x0,%edx
  14:   a1 00 00 00 00          mov    0x0,%eax
  19:   8d 04 02                lea    (%edx,%eax,1),%eax
  1c:   a3 00 00 00 00          mov    %eax,0x0
  21:   a1 00 00 00 00          mov    0x0,%eax
  26:   59                      pop    %ecx
  27:   5d                      pop    %ebp
  28:   8d 61 fc                lea    -0x4(%ecx),%esp
  2b:   c3                      ret
```



Adresses of variables and functions are not resolved (=0)!

...you don't have to understand the assembler code...

Linker and symbols: relocation table



Relocation table

- Contains information about the *relative location* of the related address in the text section for addresses initialized to "0" in the object file (*relocation offset*) and the address length (R_386_32 = 32 bit)

```
$ readelf -r foo.o

Relocation section '.rel.text' at offset 0x364 contains 4 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000010  00000801 R_386_32          00000000   a
00000015  00000901 R_386_32          00000000   b
0000001d  00000b01 R_386_32          00000004   c
00000022  00000b01 R_386_32          00000004   c
```

Linker and relocations: resolution

Offsets inside of the text segment

```
Relocation section '.rel.text' at offset 0x364 contains 4 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000010  00000801 R_386_32          00000000   a
00000015  00000901 R_386_32          00000000   b
0000001d  00000b01 R_386_32          00000004   c
00000022  00000b01 R_386_32          00000004   c
```

```
00000000 <main>:
   0:    8d 4c 24 04          lea    0x4(%esp),%ecx
   4:    83 e4 f0             and    $0xfffffff0,%esp
   7:    ff 71 fc             pushl  -0x4(%ecx)
   a:    55                   push   %ebp
   b:    89 e5                mov    %esp,%ebp
   d:    51                   push   %ecx
   e:    8b 15 00 00 00 00    mov    0x0,%edx
  14:    a1 00 00 00 00       mov    0x0,%eax
  19:    8d 04 02             lea    (%edx,%eax,1),%eax
  1c:    a3 00 00 00 00       mov    %eax,0x0
  21:    a1 00 00 00 00       mov    0x0,%eax
  26:    59                   pop    %ecx
  27:    5d                   pop    %ebp
  28:    8d 61 fc             lea    -0x4(%ecx),%esp
  2b:    c3                   ret
```

Resolved resolutions in the executable

```
$ objdump -d foo
```

foo – executable file!

```
foo:     file format elf32-i386

Disassembly of section .text:

08048374 <main>:
 8048374: 8d 4c 24 04              lea    0x4(%esp),%ecx
 8048378: 83 e4 f0                 and    $0xfffffff0,%esp
 804837b: ff 71 fc                 pushl  -0x4(%ecx)
 804837e: 55                       push   %ebp
 804837f: 89 e5                    mov    %esp,%ebp
 8048381: 51                       push   %ecx
 8048382: 8b 15 60 84 04 08        mov    0x8048460,%edx
 8048388: a1 6c 95 04 08           mov    0x804956c,%eax
 804838d: 8d 04 02                 lea    (%edx,%eax,1),%eax
 8048390: a3 78 95 04 08           mov    %eax,0x8049578
 8048395: a1 78 95 04 08           mov    0x8049578,%eax
 804839a: 59                       pop    %ecx
 804839b: 5d                       pop    %ebp
 804839c: 8d 61 fc                 lea    -0x4(%ecx),%esp
 804839f: c3                       ret
```

Addresses of variables and functions are now resolved (!=0)!

Resolved resolutions vs. "nm"

```
08048374 <main>:
 8048374: 8d 4c 24 04          lea    0x4(%esp),%ecx
 8048378: 83 e4 f0             and    $0xfffffff0,%esp
 804837b: ff 71 fc             pushl  -0x4(%ecx)
 804837e: 55                   push   %ebp
 804837f: 89 e5                mov    %esp,%ebp
 8048381: 51                   push   %ecx
 8048382: 8b 15 60 84 04 08    mov    0x8048460,%edx
 8048388: a1 6c 95 04 08       mov    0x804956c,%eax
 804838d: 8d 04 02             lea    (%edx,%eax,1),%eax
 8048390: a3 78 95 04 08       mov    %eax,0x8049578
 8048395: a1 78 95 04 08       mov    0x8049578,%eax
 804839a: 59                   pop    %ecx
 804839b: 5d                   pop    %ebp
 804839c: 8d 61 fc             lea    -0x4(%ecx),%esp
 804839f: c3                   ret
```

```
; Variable a
; ...b

; and c!
```

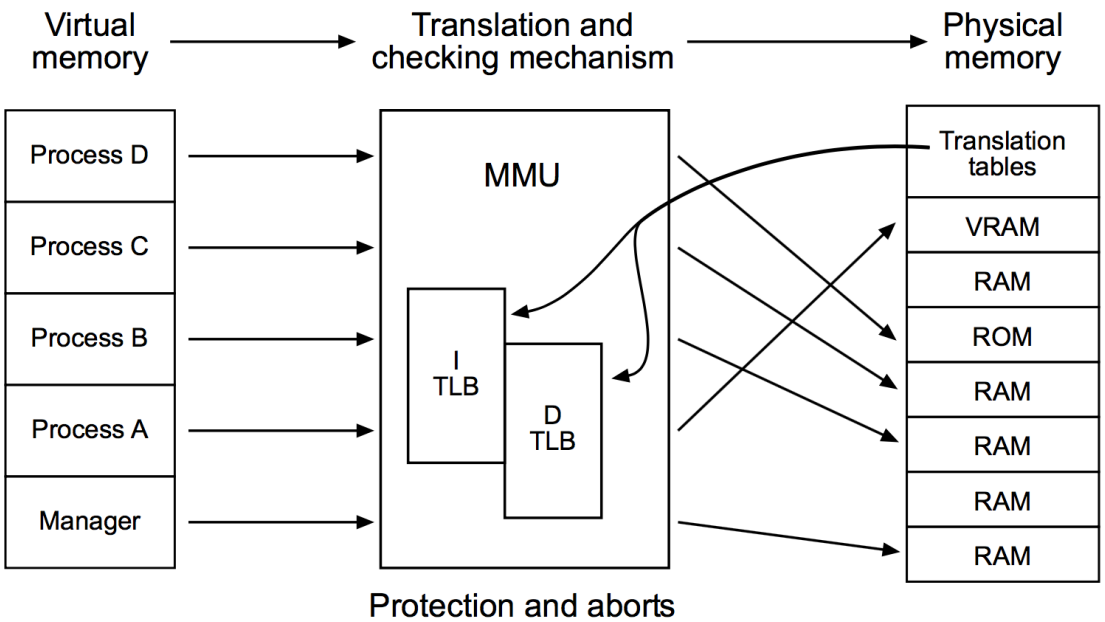
```
$ gcc -o foo foo.o
$ nm foo
[…]
08048254 T _init
080482c0 T _start
08048460 R a
0804956c D b
08049578 B c
[…]
08048374 T main
```

Assignment of addresses to variable names in executable file obtained using "nm"

Virtual memory in Unix

Linux requires a memory management unit (MMU*)

- Translates virtual addresses to physical addresses
  - **Illusion:** every process has the complete address space for its own use
  - Protection of (physical) memory from unwanted accesses
  - Granularity: "page" (e.g. 4096 bytes)

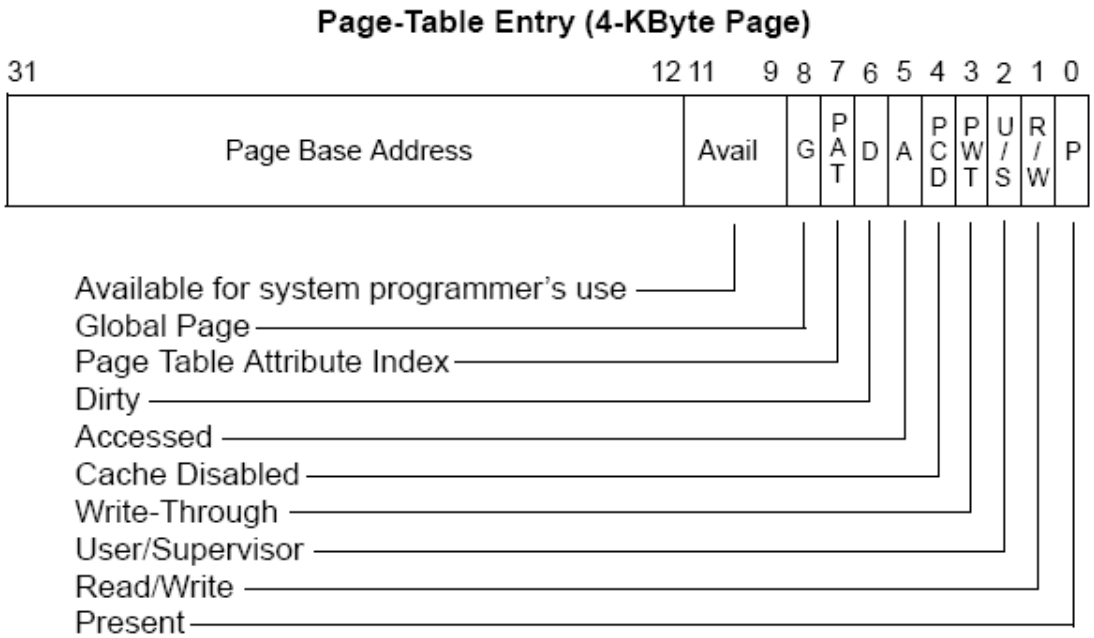


* there is a Linux version called uCLinux that works without a MMU

Virtual memory in Unix (2)

Address translation using *page tables*

- For each (mapped) *memory page* in contains the address of the related physical *page frame* (of identical size)
  - **Sparse mapping:** only map pages actually present in physical memory
  - Page table entries also contain information on *access permissions:* read/write* (and much more)

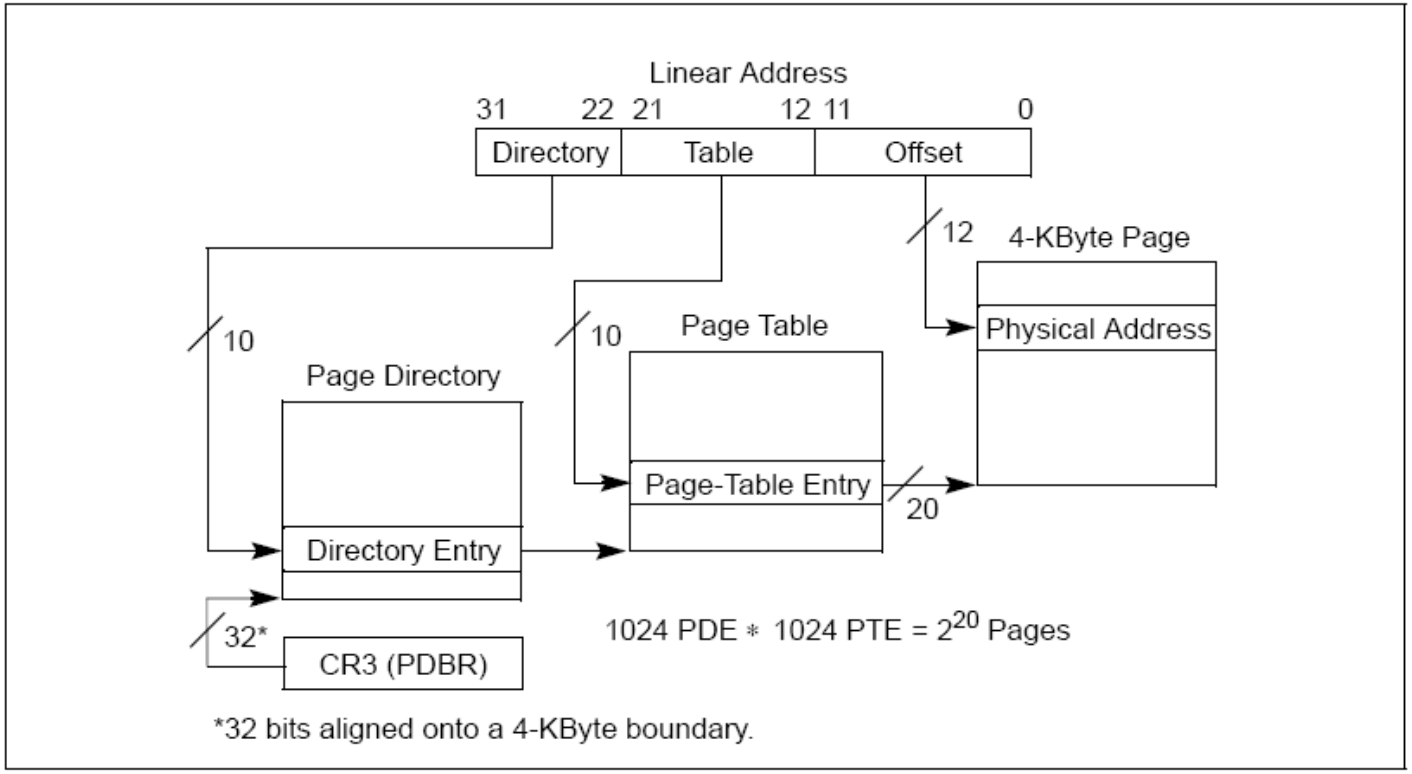


* 32 bit x86 CPUs do not provide a bit indicating the permission to *execute code* in a page!
➛ always permitted!

Virtual memory in Unix (3)

Structure of the page table

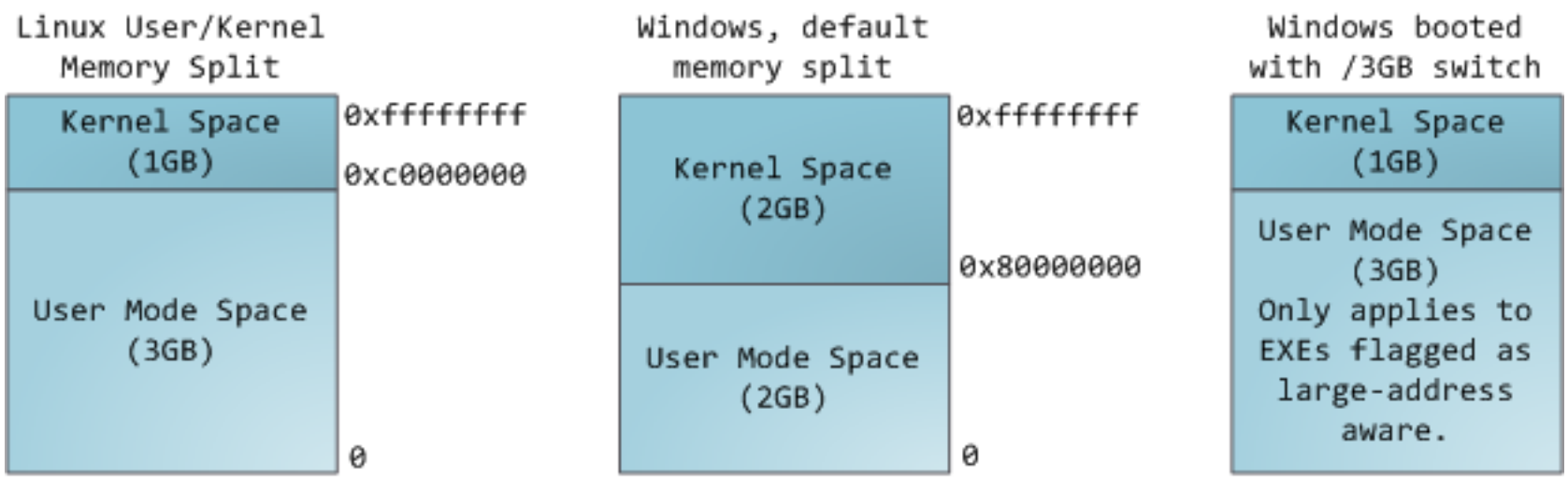
- The page table is a data structure stored in RAM
- Small excerpts are cached in the *Translation Lookaside Buffer (TLB)*



Process virtual memory layout (1)

Structure of the virtual address space (here: 32 bit = 4 GB)

- User Mode Space = process address space, separate per process
  - Every process has its own page table
  - Kernel mapped into upper part (25/50%) **of every** process address space (not on macOS)



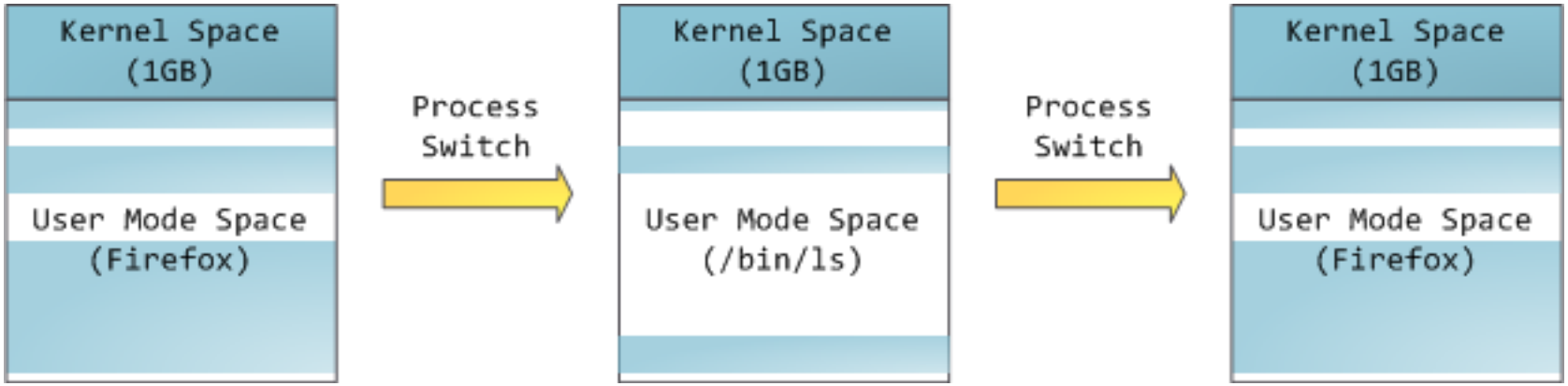
http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/

Process virtual memory layout (2)

Every process has its own page table

- Switched by the kernel when processes are switched
- *The same k*ernel address space (upper GB of virtual addresses) is *always mapped* into all process address spaces (on 32 bit x86)*

* this has changed with mitigations for Spectre side channel attacks…



http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/

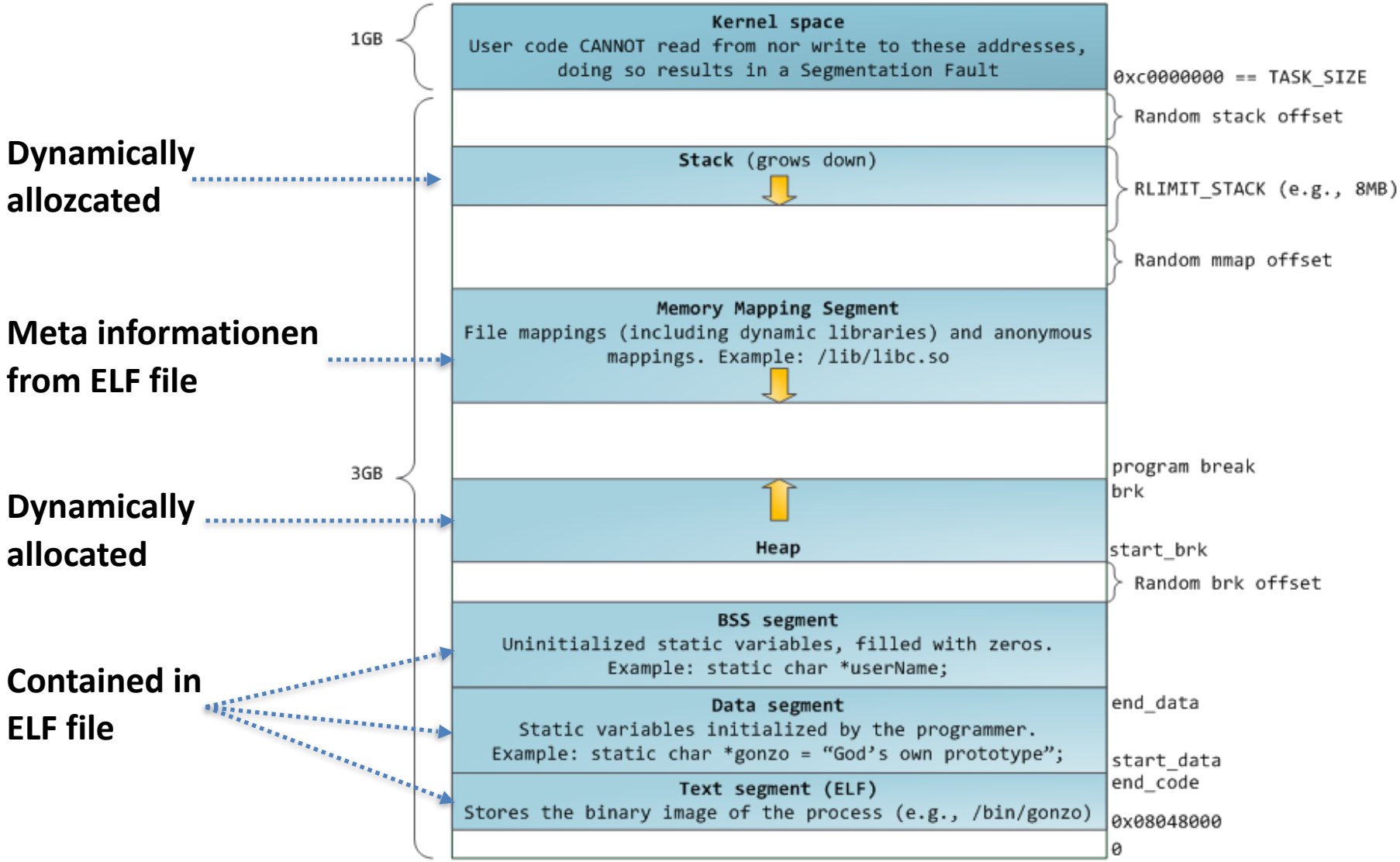
Process memory layout (1)

Transition from *program* to *process*

- Definition:

  "Process = program in execution"
- Process = **run time context** of a program
- Loading the program into RAM
  - Assign *virtual* address for program sections
  - In addition, reserve address spaces for dynamically growing areas
    - *Heap and stack*
  - Address space above 3GB (=0xC000 0000 – on 32 bit x86) not read/write/executable for the process
    - kernel memory is mapped there!

Process memory layout (2)



Defining the process memory layout (1)

Linker uses a *linker script*

- Defines address space regions for ELF sections
- Additional: entry points, segment orders, alignment, ...

Simple linker script:

```
SECTIONS
{
   . = 0x10000;
   .text : { *(.text) }
   . = 0x8000000;
   .data : { *(.data) }
   .bss : { *(.bss) }
}
```

Start at *virtual address* 0x10000:

- text segment

Start at *virtual address* 0x8000 0000:

- first: .data
- directly after: .bss

Defining the process memory layout (2)

Check: Does the linker skript work?

- Print symbols using *nm*:

```
$ gcc -m32 -c foo.c
$ ld -m elf_i386 -T foo.ld -o foo
foo.o
$ nm foo
0001002c R a
08000000 D b
08000004 B c
00010000 T main
```

Linker script *foo.ld*:

```
SECTIONS
{
  . = 0x10000;
  .text : { *(.text) }
  . = 0x8000000;
  .data : { *(.data) }
  .bss : { *(.bss) }
}
```

Example program foo.c:

```
const int a = 42;
int b = 23;
int c;

int main(void) {
  c = a + b;
  return c;
}
```

Defining the process memory layout (3)

...section .rodata not defined in linker script: automatically allocated!

```
$ gcc -m32 -c foo.c
$ ld -m elf_i386 -T foo.ld -o fox
foo.o
$ nm fox
08000000 R a
08000004 D b
08000008 B c
00010000 T main
```

...works!

Linker scripts can do much more – reorder object files, address arithmetics, alignment, split between ROM/RAM areas, ...

Linker script *foo.ld*:

```
SECTIONS
{
  . = 0x10000;
  .text : { *(.text) }
  . = 0x8000000;
  .rodata : { *(.rodata) }
  .data : { *(.data) }
  .bss : { *(.bss) }
}
```

Example program foo.c:

```
const int a = 42;
int b = 23;
int c;

int main(void) {
  c = a + b;
  return c;
}
```

Defining the process memory layout (4)

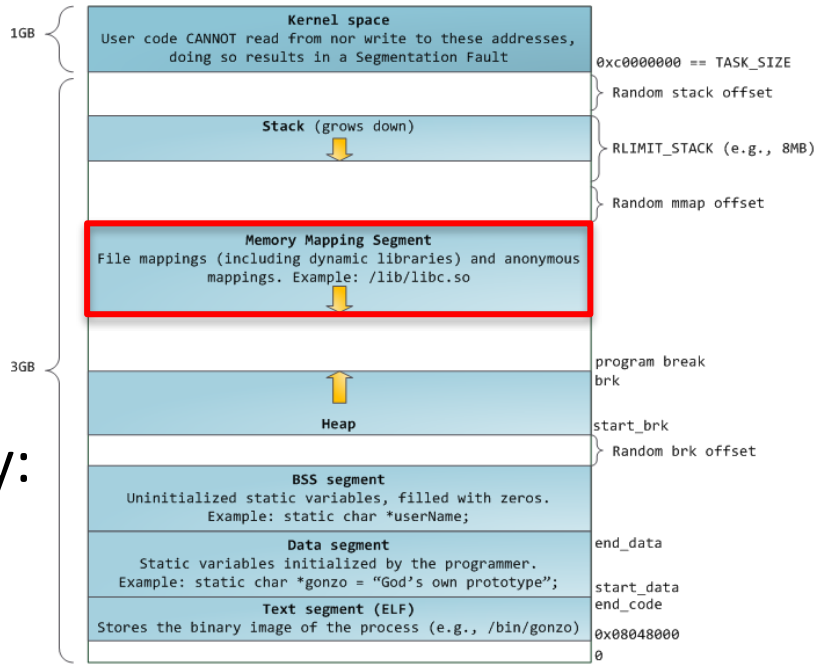
Linker script *foo.ld*:

```
SECTIONS
{
  . = 0x10000;
  .text : { *(.text) }
  . = 0x8000000;
  .rodata : { *(.rodata) }
  .data : { *(.data) }
  .bss : { *(.bss) }
}
```

Check the layout using readelf:

```
$ readelf -l fox

Elf file type is EXEC (Executable file)
Entry point 0x10000
There are 3 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x001000 0x00010000 0x00010000 0x00030 0x00030 R E 0x1000
  LOAD           0x002000 0x08000000 0x08000000 0x00004 0x00008 RW  0x1000
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4

 Section to Segment mapping:
  Segment Sections...
   00     .text
   01     .rodata .data .bss
   02
```

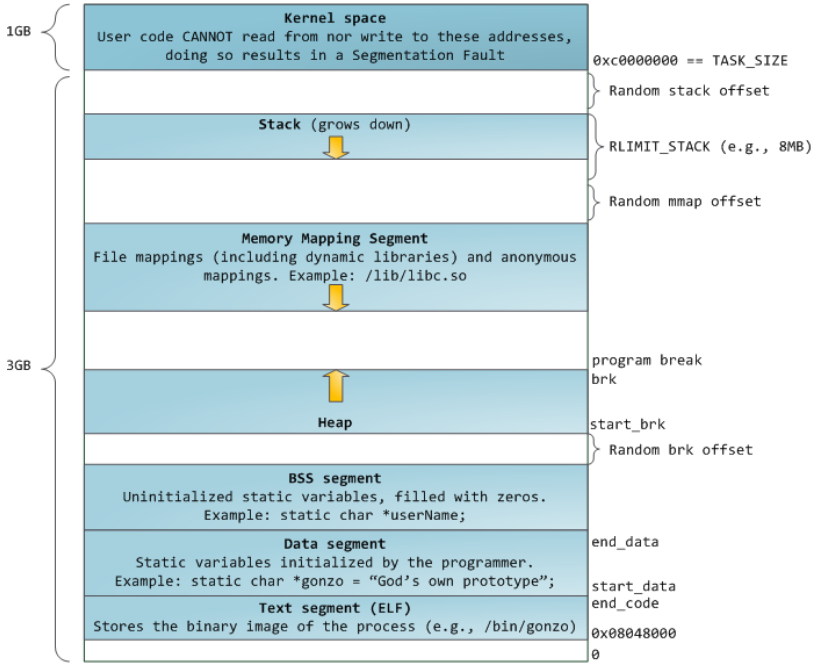
Shared libraries (1)



Programs (usually) don't stand alone

- This is possible, but requires a lot of work...
- Libraries contain useful functionality:
  - libc – C standard functions
  - libm – mathematical functions
  - libX11 – X11 window system functions
- Many programs use the same libraries
  - With *static linking*: *copy* of library code contained in every executable progrsam ➙ *waste of disk space*
  - Copies of the library code are allocated in memory for each process ➙ *waste of RAM*

Shared libraries (2)



Shared libraries are ELF files

- Usually contain no main function
- Functions/symbols not contained in the ELF file of a program are resolved in the shared libraries
- Name resolution is costly
  - Shared libraries use hash tables

Dynamic linker ld.so

Which shared libraries are referenced by an ELF file?

```
$ gcc -m32 -o foo foo.o
$ ldd foo
        linux-gate.so.1 =>  (0xb7701000)
        libc.so.6 => /lib32/libc.so.6 (0xb759f000)
        /lib/ld-linux.so.2 (0xb7702000)
```

"ldd" (list dynamic dependencies) prints list of shared libraries

"on top" of the process address space: almost at 0xC000 0000

File name of shared library "libc"

File name of the dynamic loader to be used

ld.so (here: /lib/ld-linux.so.2 – for ELF)…

- searches and loads the shared libraries required by a program
- prepares the program for execution
- and starts it

Special shared libraries

What is linux-gate.so.1?

```
$ gcc -m32 -o foo foo.o
$ ldd foo
        linux-gate.so.1 =>  (0xb7701000)
        libc.so.6 => /lib32/libc.so.6
(0xb759f000)
        /lib/ld-linux.so.2 (0xb7702000)
```

...only for the sake of completeness...

There is no file for **linux-gate.so.1**!

- Virtual shared library, mapped into process address space by the kernel
- Provides portable mechanisms for system calls

Details at http://www.trilithium.com/johan/2005/08/linux-gate/

Runtime process memory layout (1)

Virtual file system `/proc`

- Contains kernel information on system and process state, e.g.:
  - Memory layout: `/proc/meminfo`
  - System runtime: `/proc/uptime`
  - Kernel version: `/proc/version`
- Information about running processes in `/proc/[pid]/`, e.g..
  - `/proc/[pid]/cmdline`: Command line
  - `/proc/[pid]/sched`: Scheduling statistics
  - `/proc/[pid]/stack`: Current stack contents (call hierarchy)
  - `/proc/[pid]/maps`: **Allocation of virtual memory**

Runtime process memory layout (2)

/proc/[pid]/maps

- Example: retrieve memory layout for prozess „foo“

```
$ ps ax | grep foo
10323
```

The pid is dynamically allocated when the process is started

Which process ID does foo have here? **10323**

```
$ cat /proc/10323/maps
08048000-08049000 r-xp 00000000 90:5a 10520727 /home/me/f/foo
08049000-0804a000 rw-p 00000000 90:5a 10520727 /home/me/f/foo
b75f8000-b75f9000 rw-p 00000000 00:00 0
b75f9000-b7745000 r-xp 00000000 90:5a 8529693  /emul/ia32-linux/lib/libc-2.7.so
b7745000-b7746000 r--p 0014c000 90:5a 8529693  /emul/ia32-linux/lib/libc-2.7.so
b7746000-b7748000 rw-p 0014d000 90:5a 8529693  /emul/ia32-linux/lib/libc-2.7.so
b7748000-b774b000 rw-p 00000000 00:00 0
b7758000-b775b000 rw-p 00000000 00:00 0
b775b000-b775c000 r-xp 00000000 00:00 0        [vdso]
b775c000-b7779000 r-xp 00000000 90:5a 8529726  /emul/ia32-linux/lib/ld-2.7.so
b7779000-b777b000 rw-p 0001c000 90:5a 8529726  /emul/ia32-linux/lib/ld-2.7.so
bf9ee000-bfa03000 rw-p 00000000 00:00 0        [stack]
```

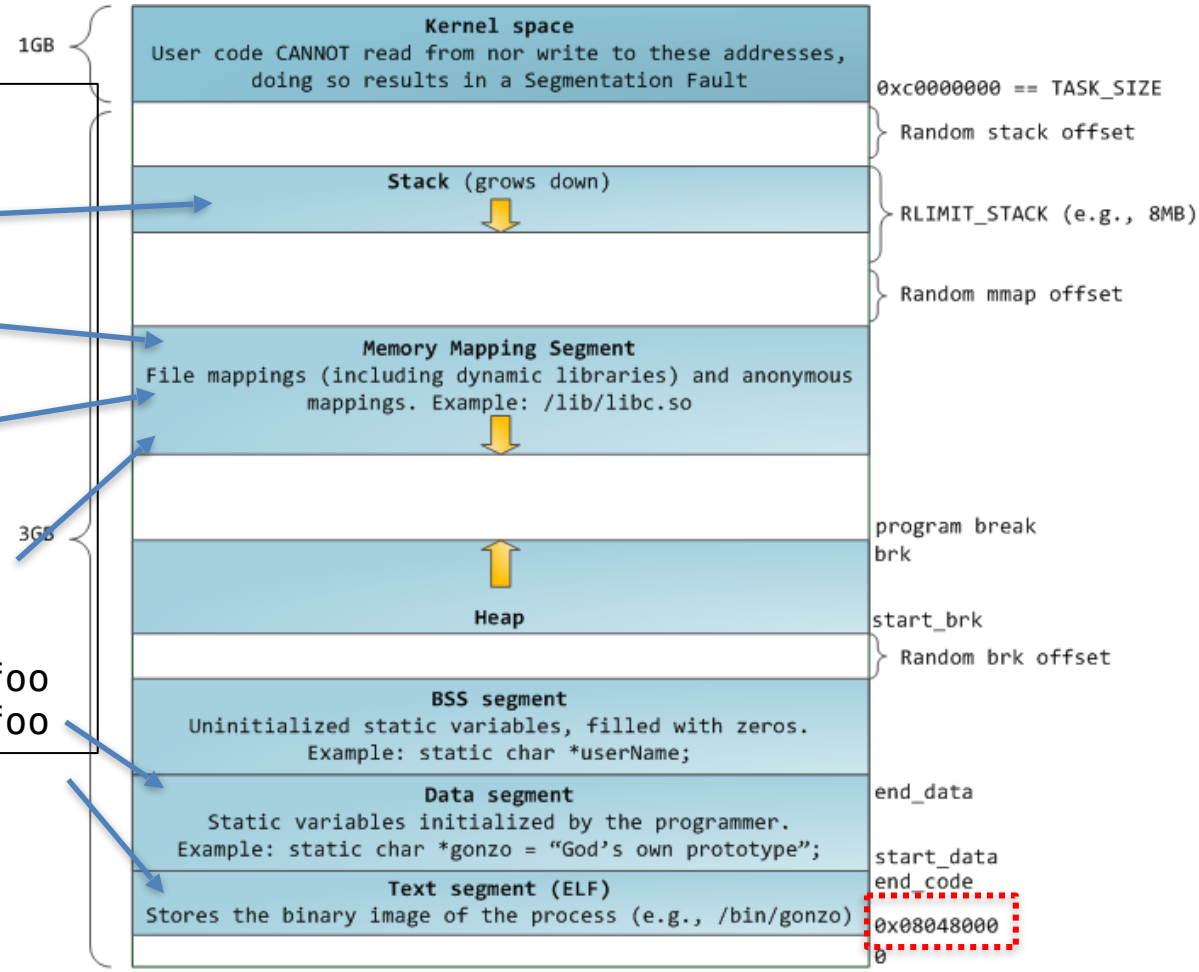
The program itself

libc

linux-gate.so

Runtime process memory layout (3)

```
...sorted and shortened…

bf9ee000-bfa03000 [stack]

b7779000-b777b000 ld-2.7.so
b775c000-b7779000 ld-2.7.so

b775b000-b775c000 [vdso]

b7746000-b7748000 libc-2.7.so
b7745000-b7746000 libc-2.7.so
b75f9000-b7745000 libc-2.7.so

08049000-0804a000 /home/me/f/foo
08048000-08049000 /home/me/f/foo
```



Loading: from ELF to memory

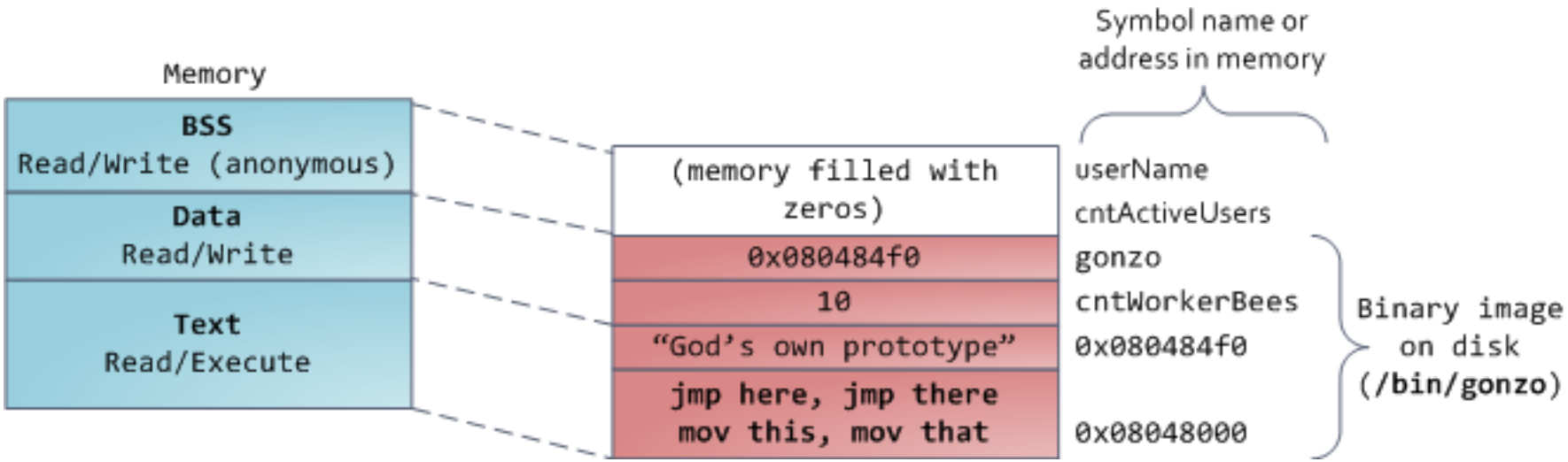


Figure out memory layout from C (1)

Global variables predefined in crt0 (via linker script):

etext.c

```
#include <stdio.h>

extern char __executable_start;
extern char __etext;

int main(void)
{
  printf("0x%lx\n", (unsigned long)&__executable_start);
  printf("0x%lx\n", (unsigned long)&__etext);
  return 0;
}
```

_etext = “end of .text section”

Output:

```
$ gcc -m32 -o etext etext.c && ./etext
0x8048000
0x80484a8
```

Figure out memory layout from C (2)

Which variables are available?

```
$ gcc -m32 -o etext -Wl,--verbose etext.c && ./etext
```

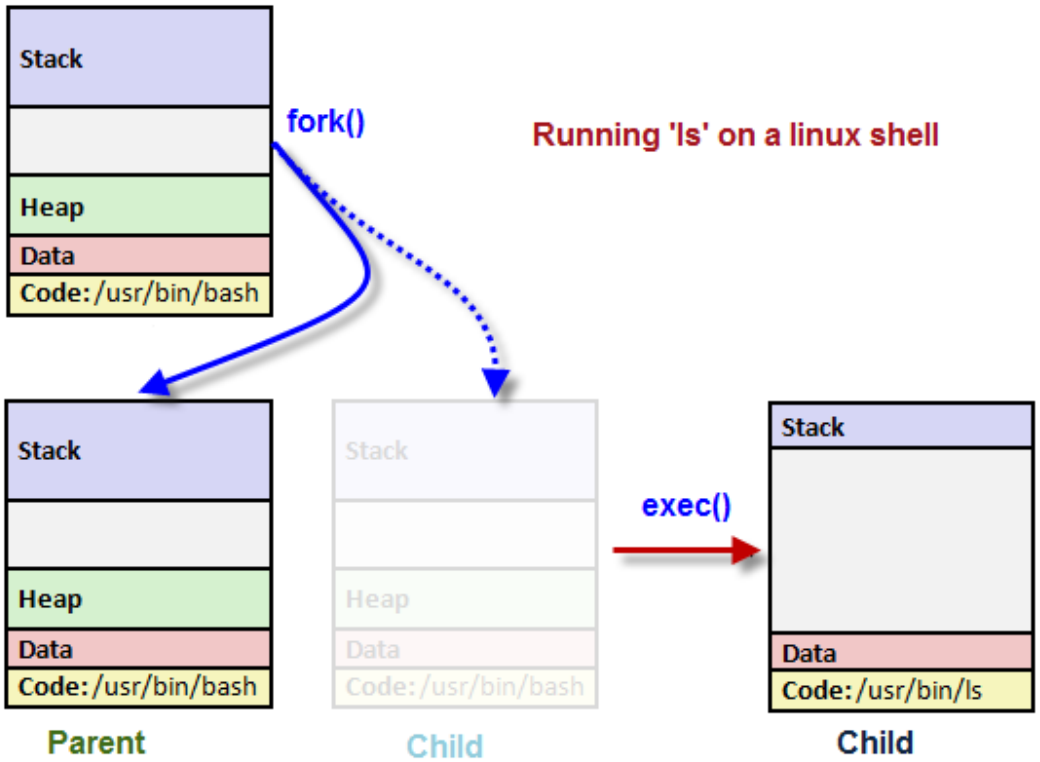
`-Wl,xxx` passes parameter "xxx" to the linker!

```
... 250 lines of output (demo) ...
```

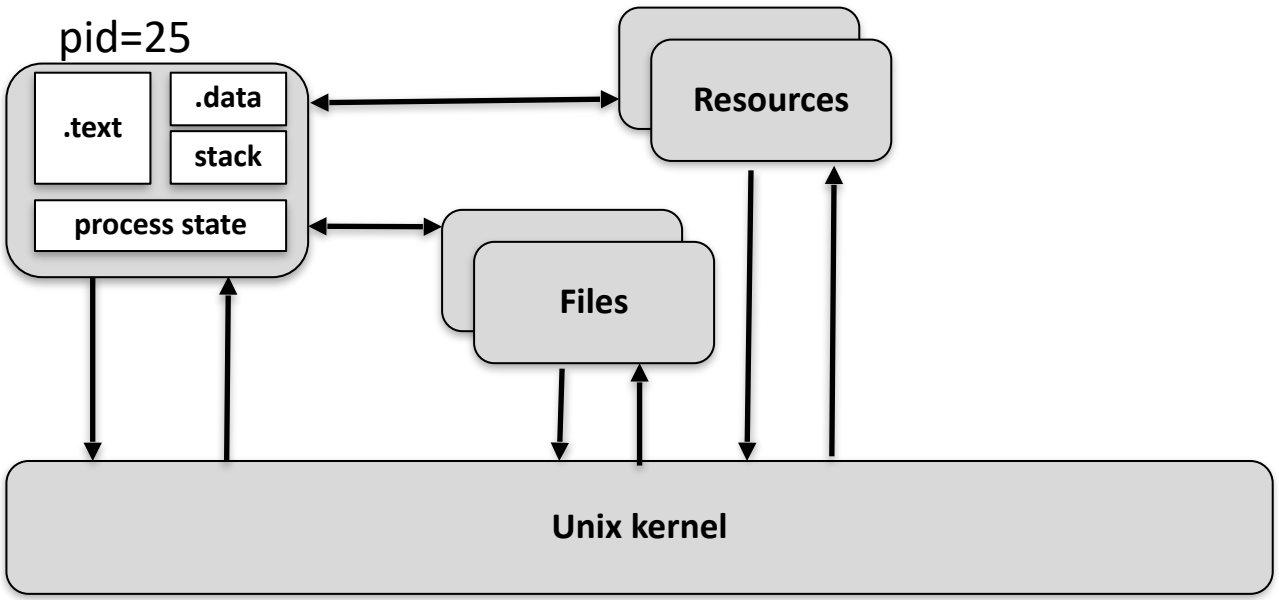
Unix process creation

Two necessary steps:

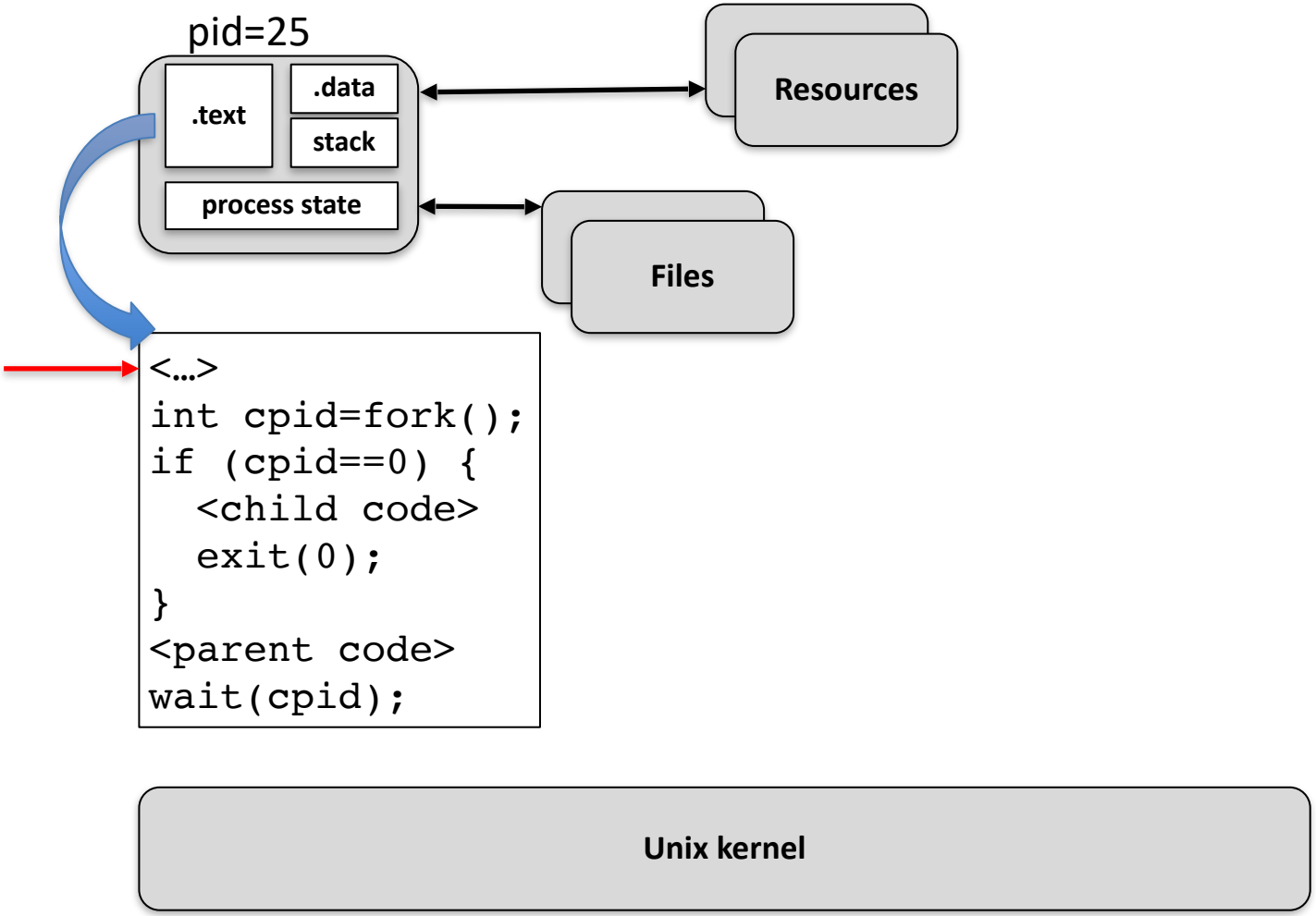
- A process creates identical copy ("child process") of itself
  - system call `fork()`
  - creates copy of the address space
- Child process over-writes its own address space with code and data of another program:
  - system call `exec()`



fork in detail (1)

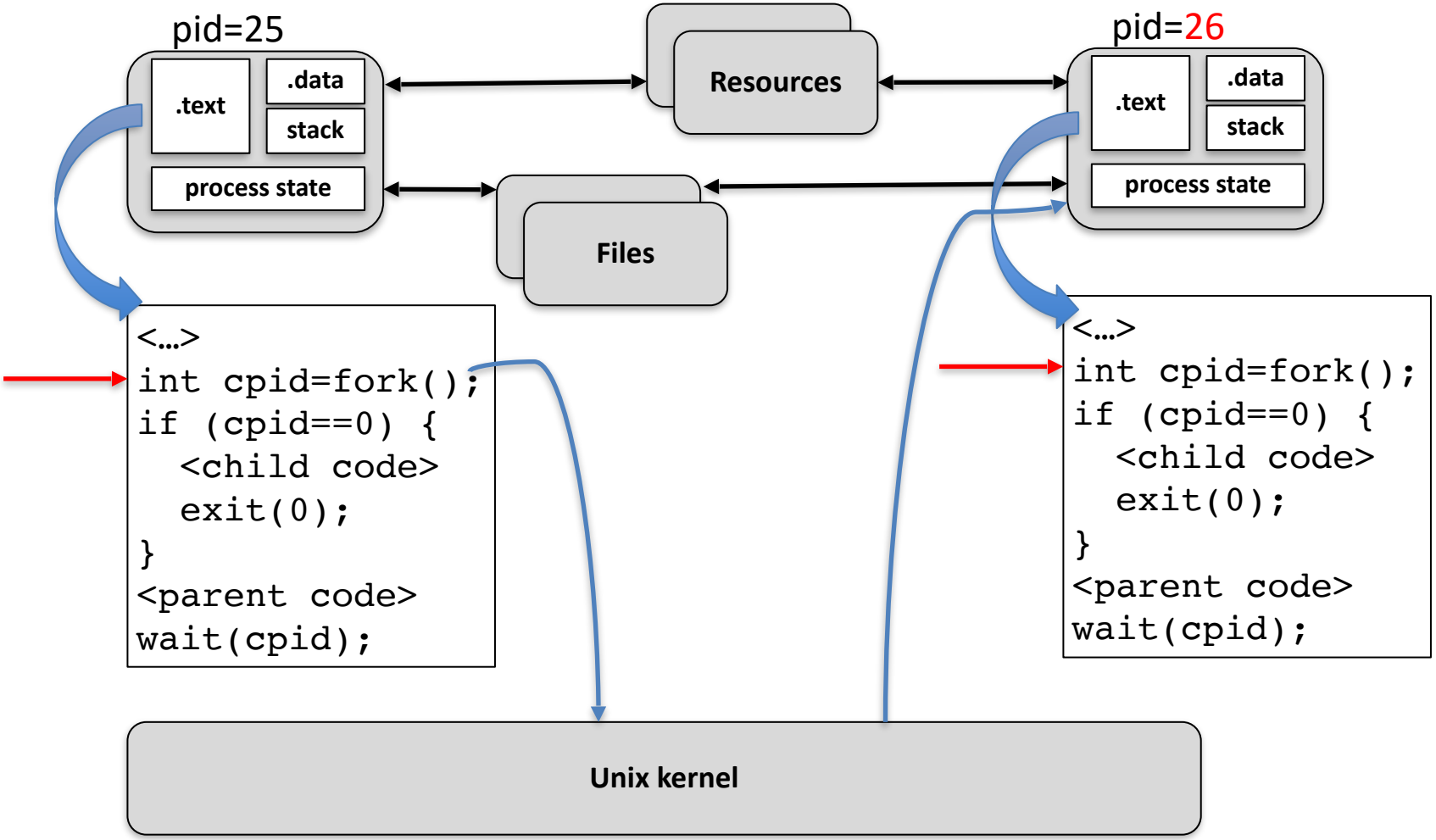


fork in detail (2)



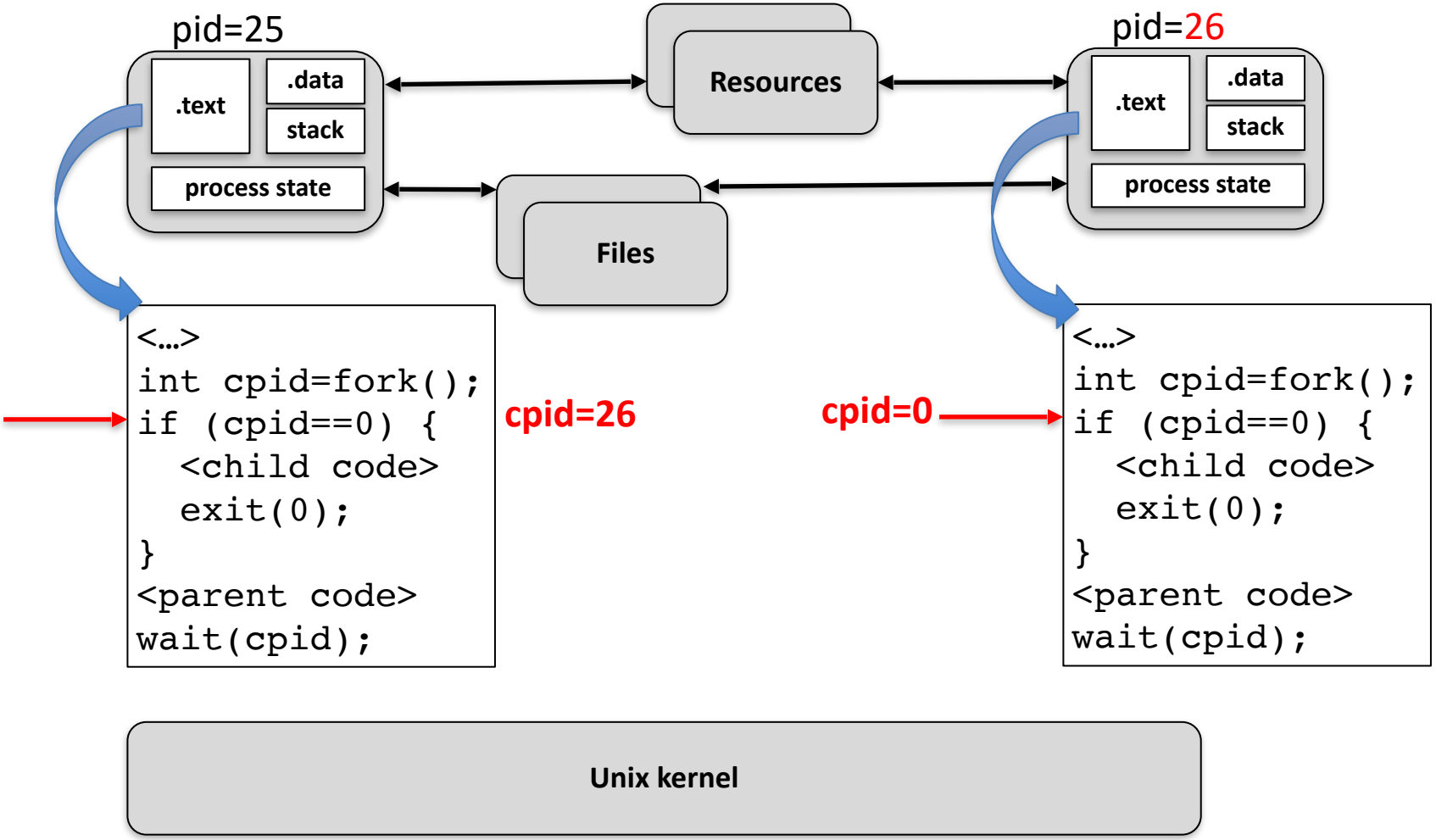
fork in detail (3)

pid26 is a copy of the process with pid25, both are in fork()!



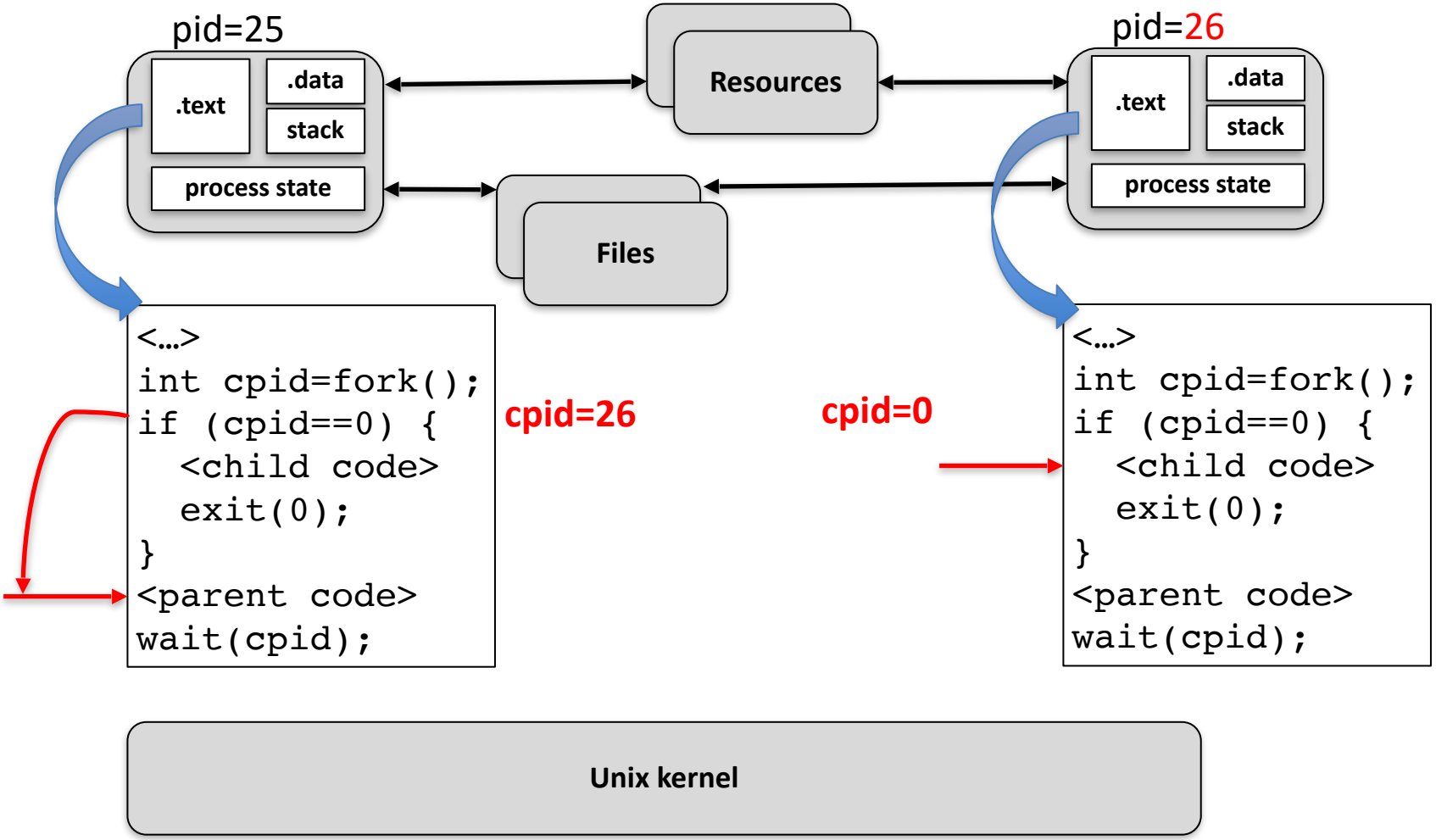
fork in detail (4)

pid25 and pid26 both return from fork.
Difference: *return value!*



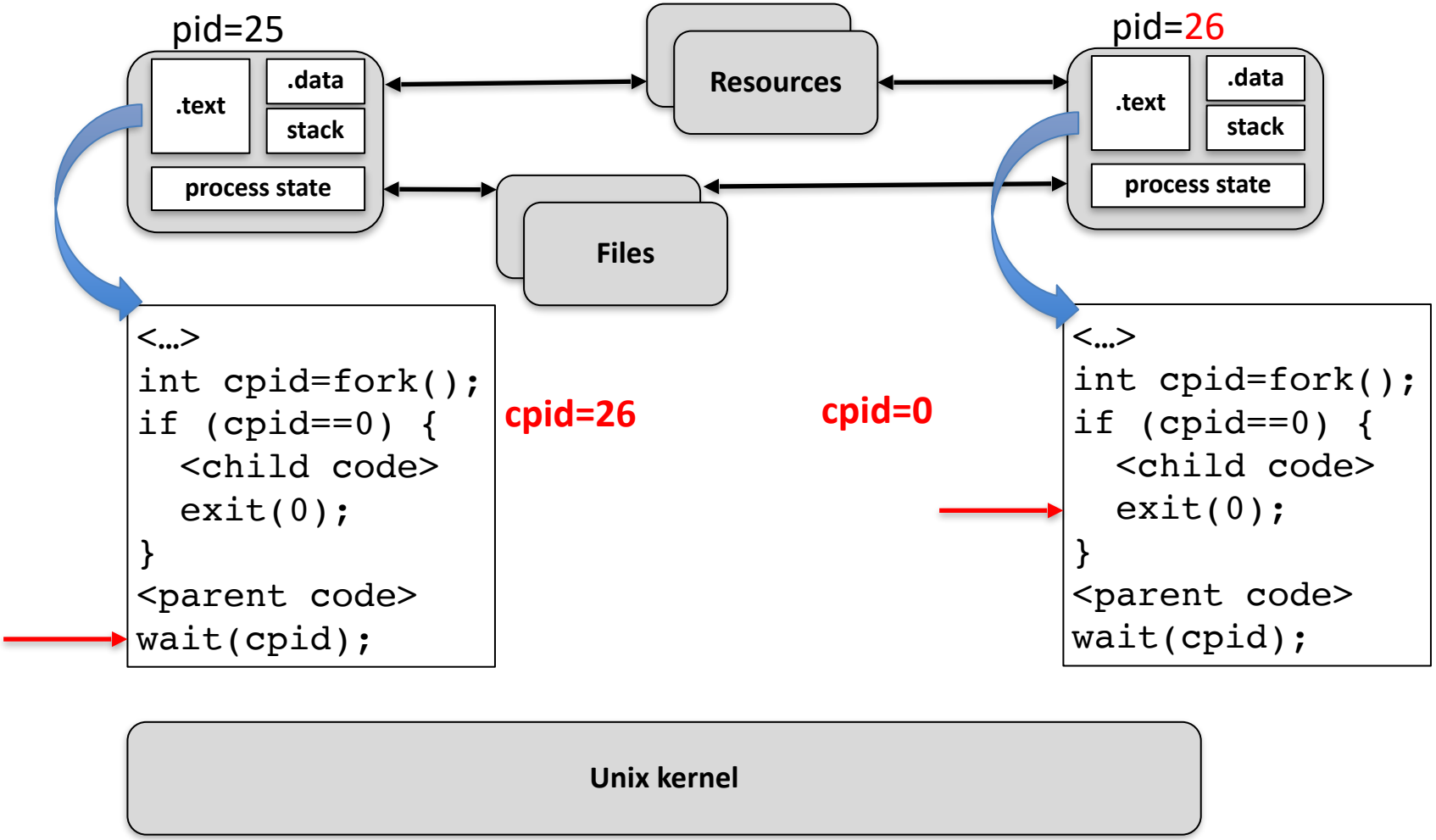
fork in detail (5)

pid25 skips the if() block whereas pid26 executes it



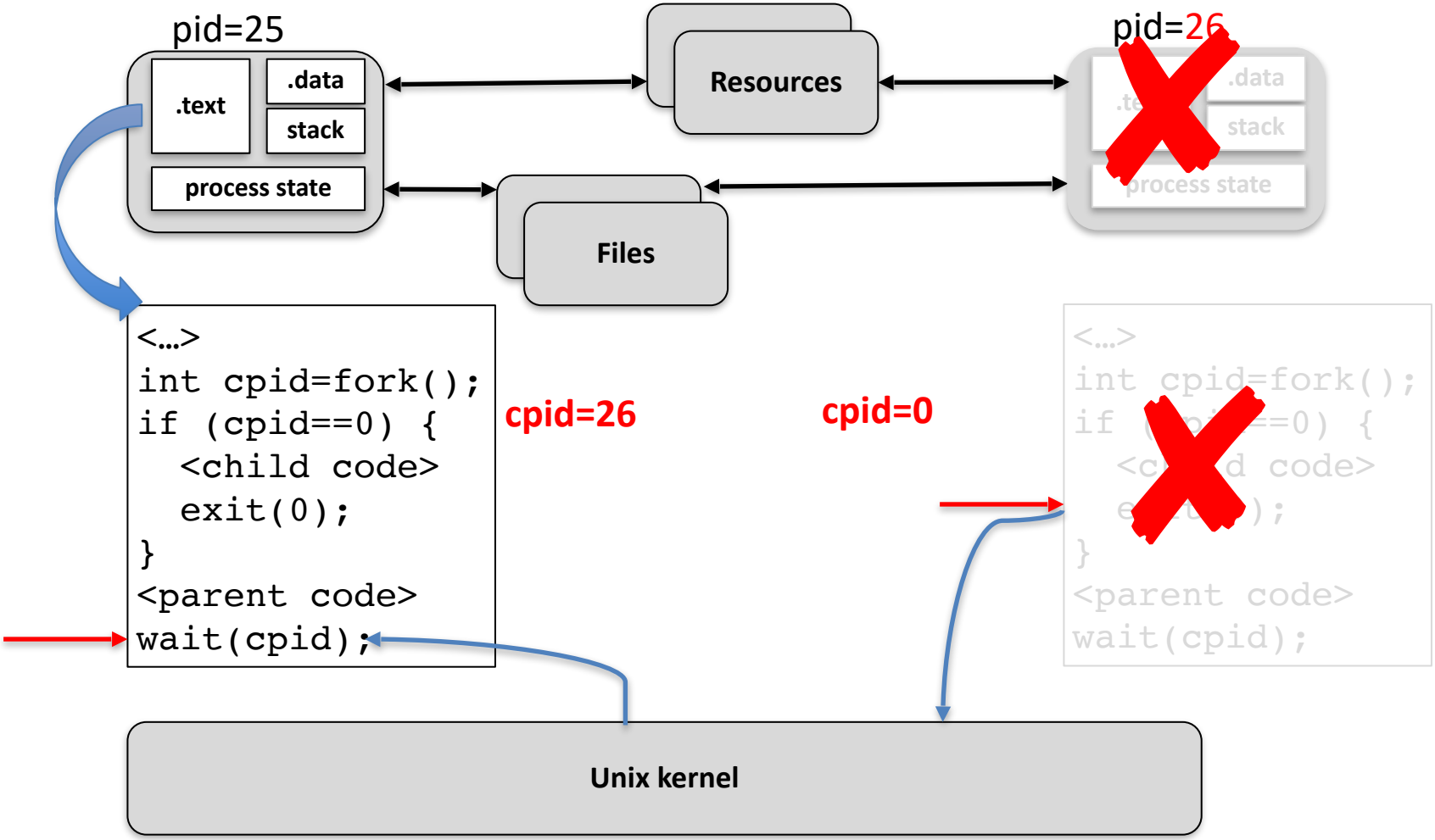
fork in detail (6)

pid25 waits for termination of pid26, pid26 executes exit(0) and terminates



fork in detail (7)

pid26 is terminated and removed from memory, pid25 stops waiting for the termination of pid26



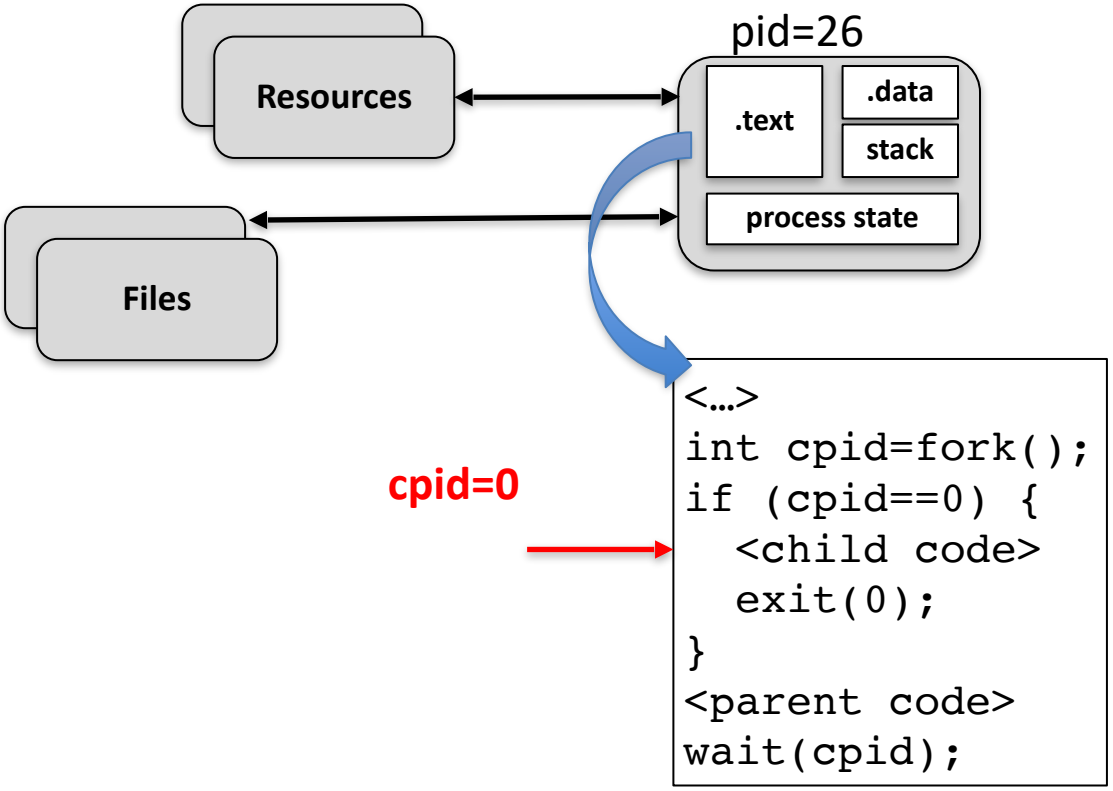
exec functionality

Replace the memory contents of the current process with the sections of another executable file (parameter of exec)

- Only method to start *new* programs in Unix
  - ...since *fork()* only creates a copy of the calling process
- Kernel opens executable file
- ELF loader loads text and data segments
- ld.so loads required shared libraries
- Stack, heap, BSS are newly created
- New program is started at its *entry point*

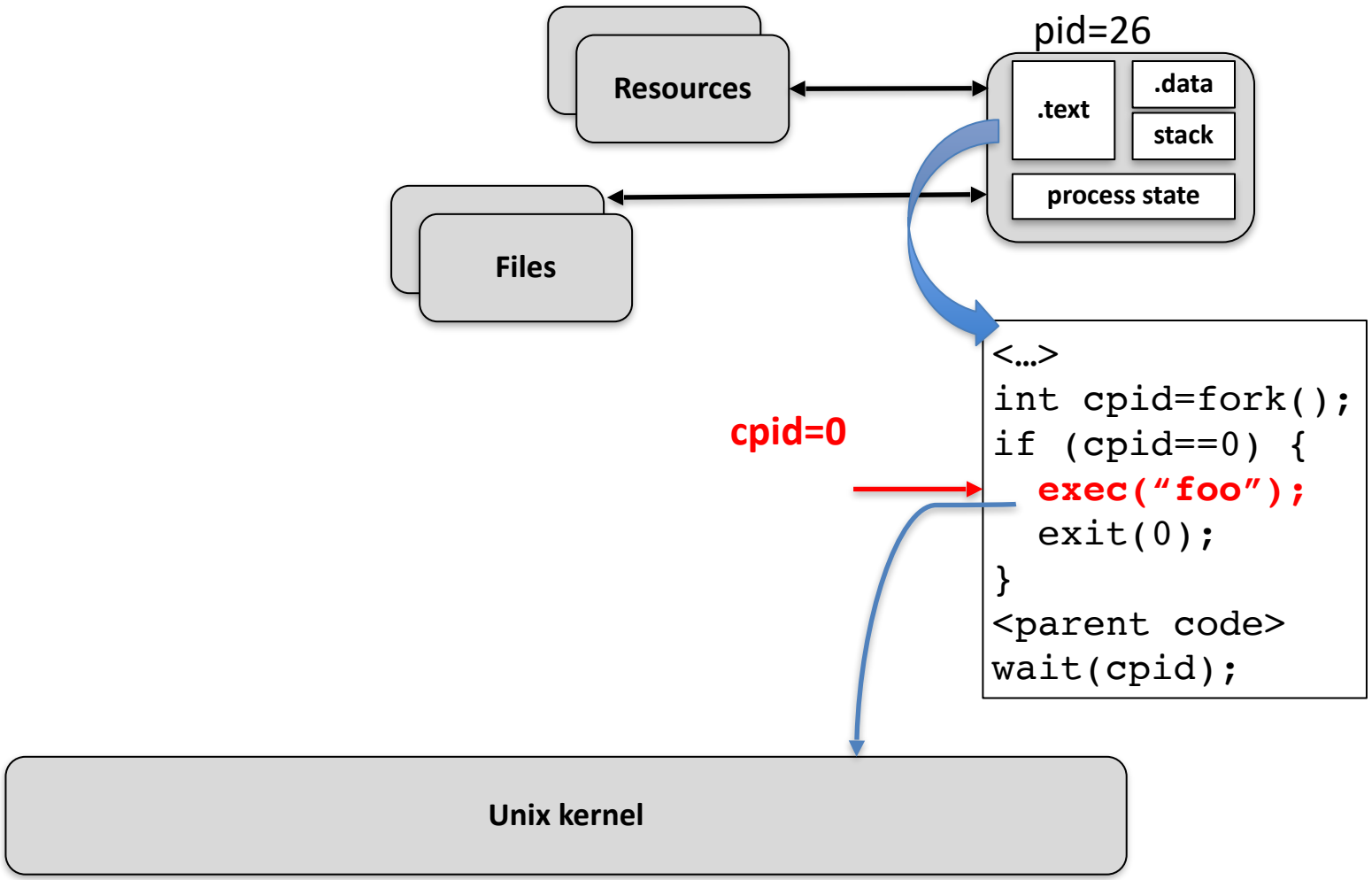
exec in detail (1)

Child process executes its if() block



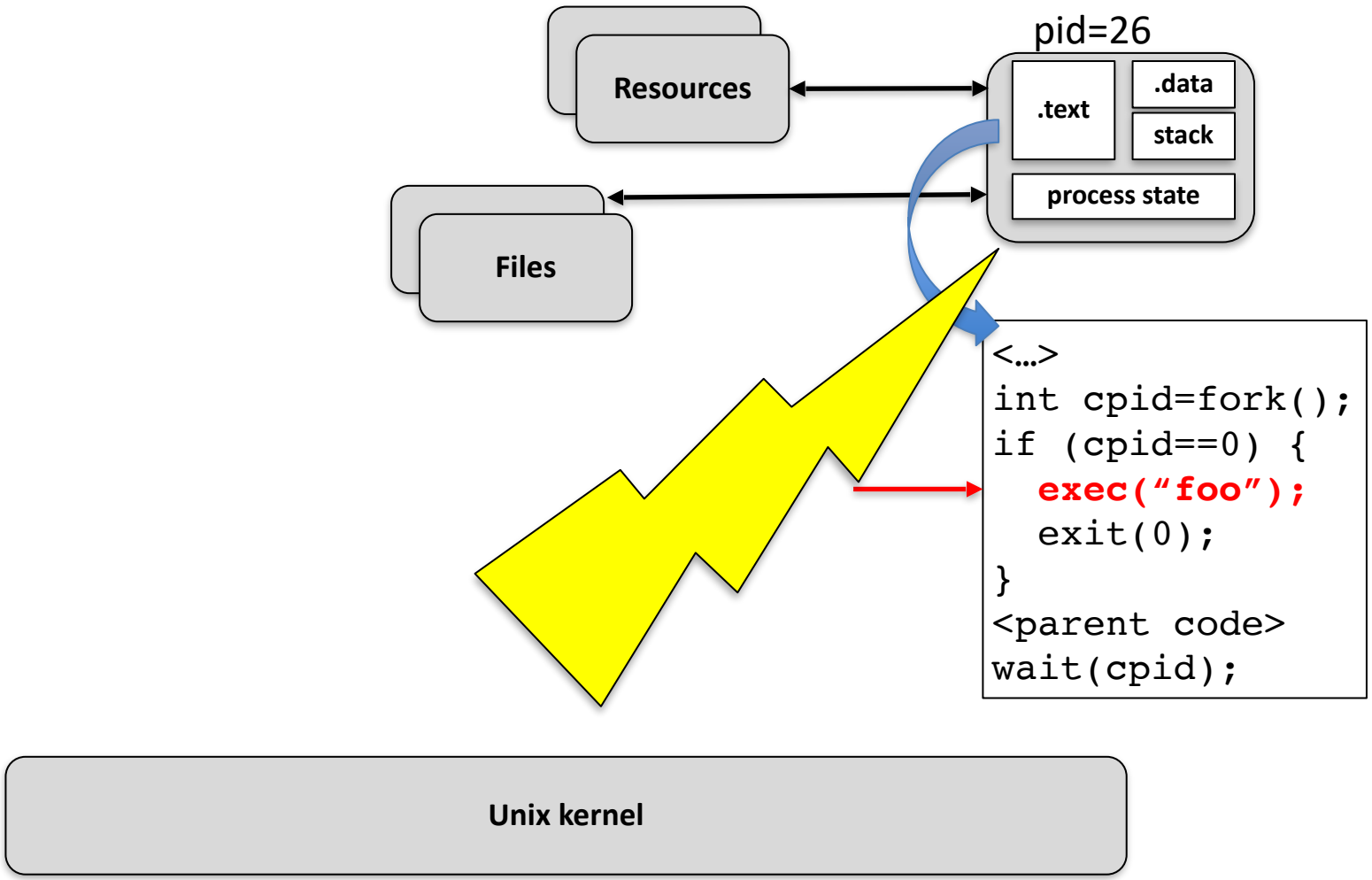
exec in detail (2)

System call exec() is called in the if-Block



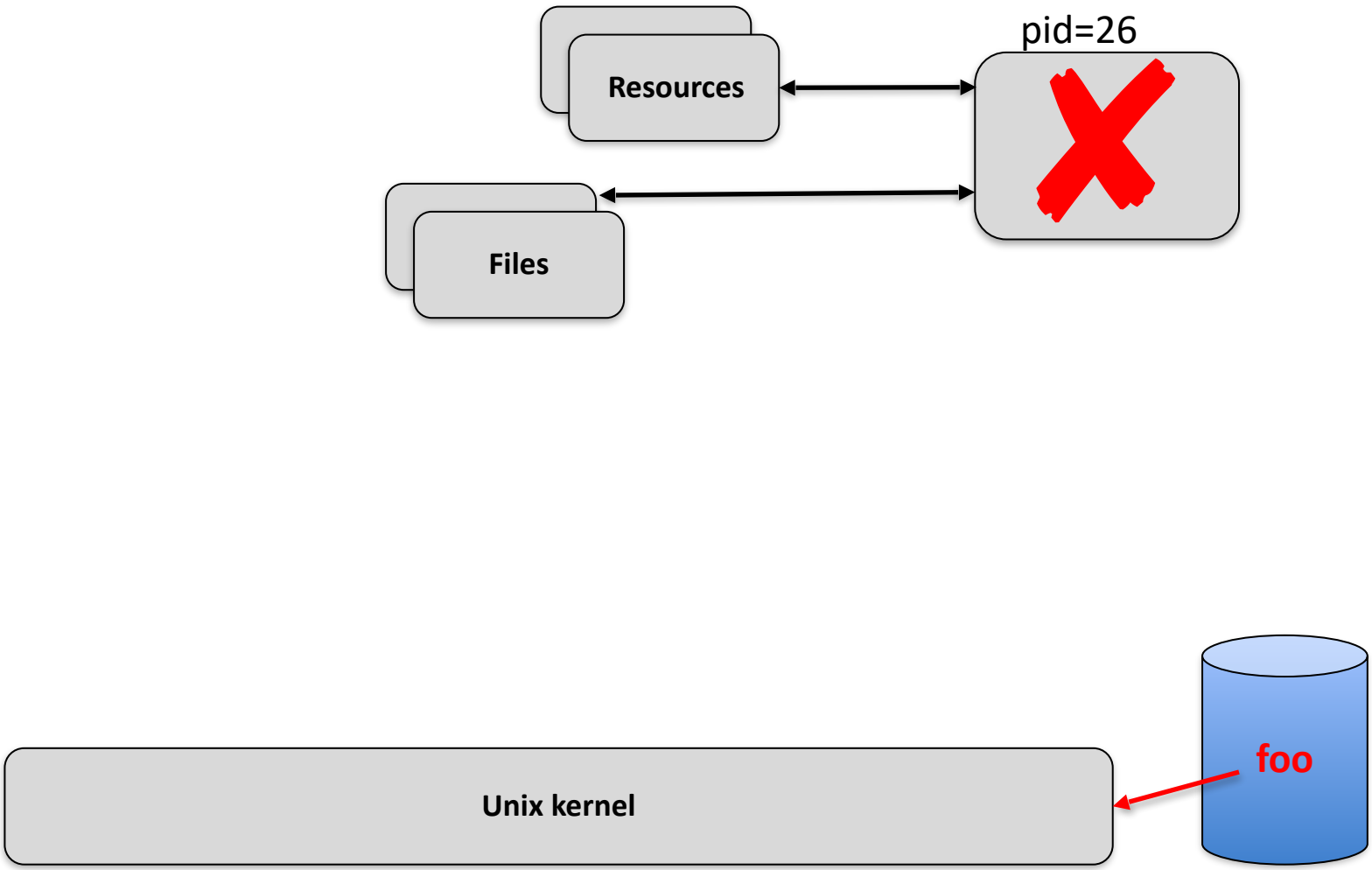
exec in detail (3)

Kernel "removes" memory content of pid26



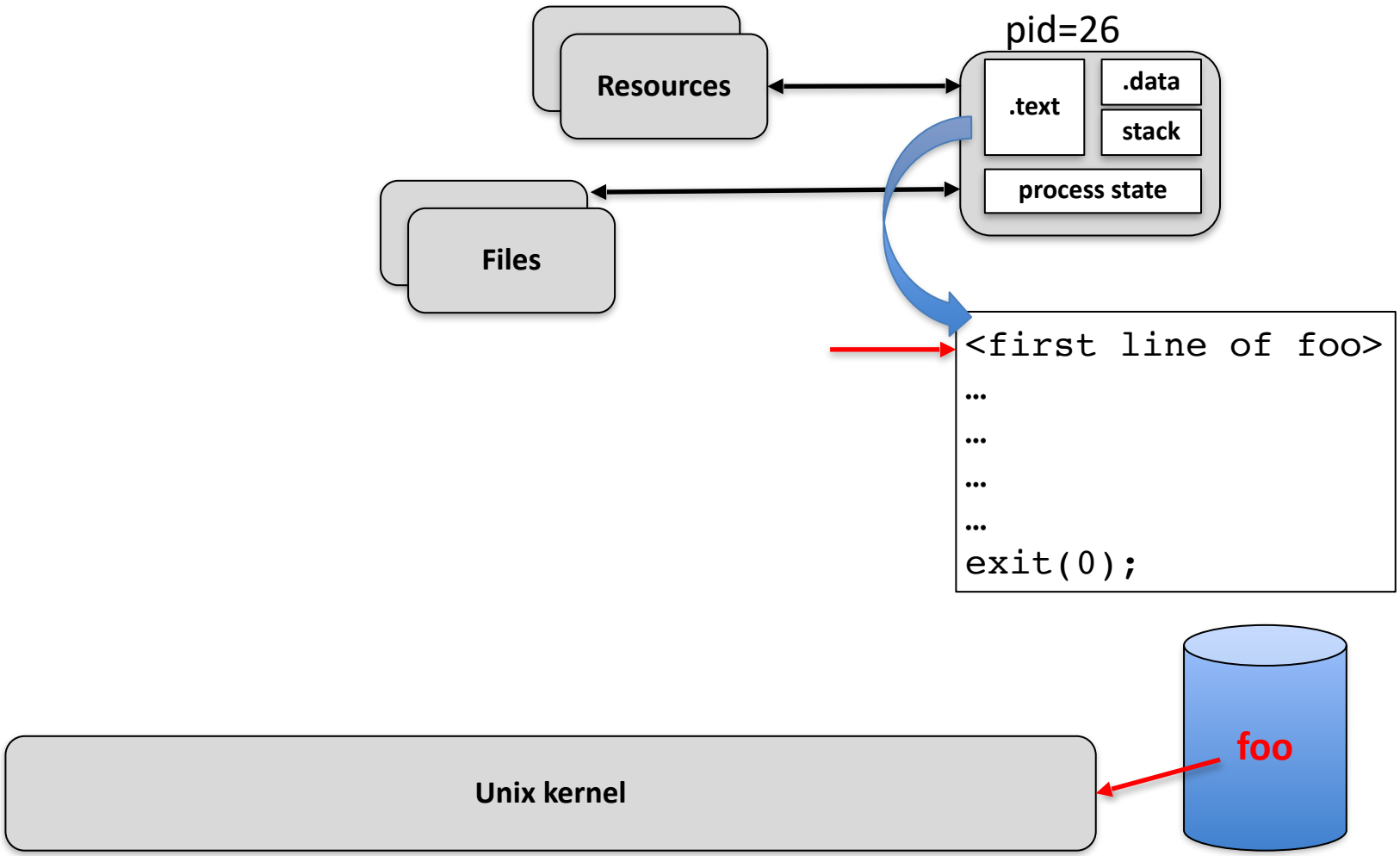
exec in detail (4)

Kernel opens and loads executsable file “foo” from the file system



exec in detail (5)

Memory of pid26 replacted by segments of "foo", starts "from the beginning"



Starting the new program after exec

Where does process created using *fork()* start?

- It is an (almost) identical copy of the parent process
- Continues afte the return from the *fork()* call

Where does a newly loaded (exec'd) program start?

- No more previous program code available
- Start "from the beginning"…
- …so, from main()? ➙ ***no!***

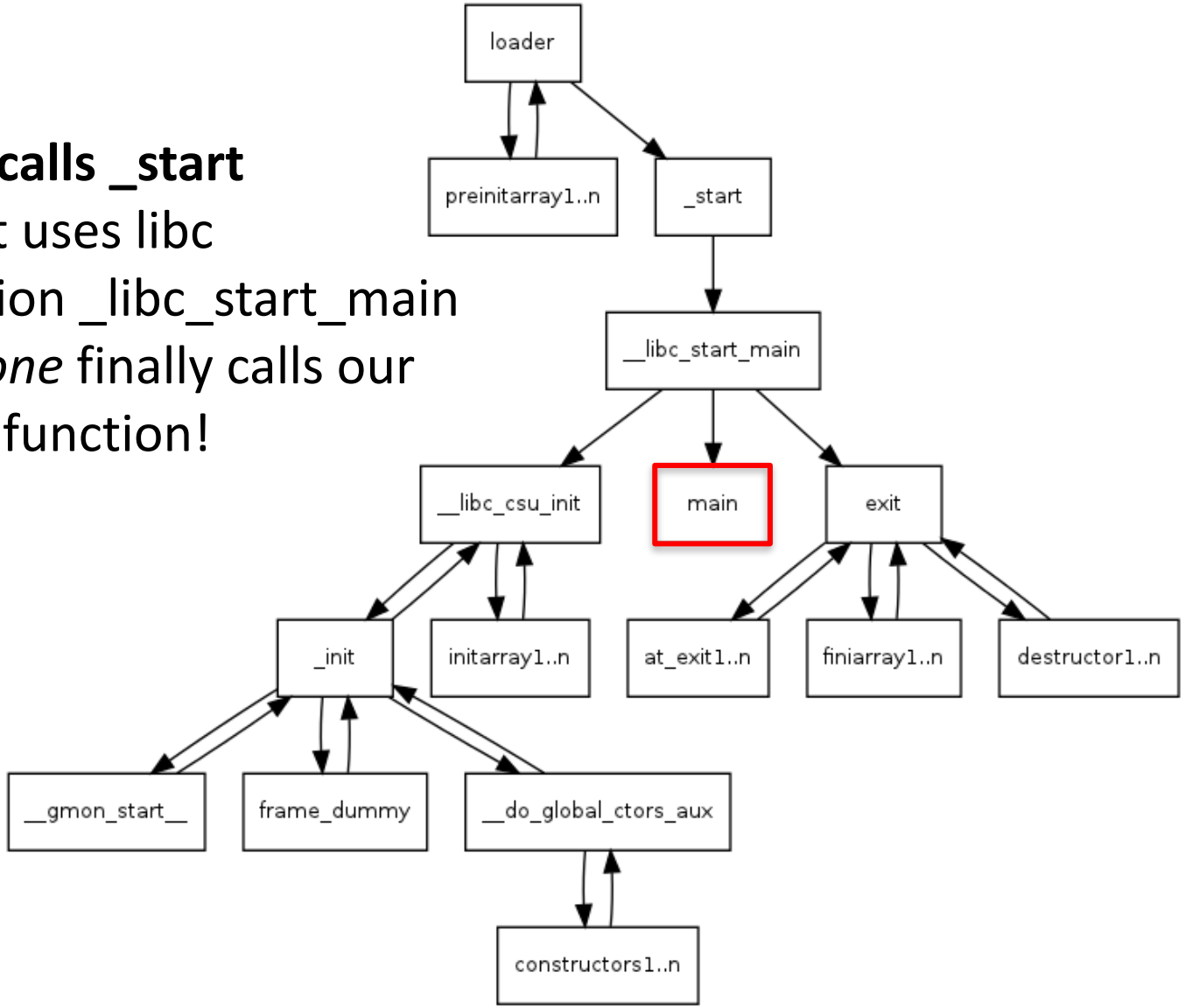
Before main is called, things have to be initialized

- e.g. set the contents of the .bss segment to zero

Program startup overview

Loader calls _start

- _start uses libc function _libc_start_main
- *This one* finally calls our main function!



A very simple C program

...just an empty main()

- In the disassembler (excerpts):

```
$ 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
```

Addresses of the instructions

Opcodes

Disassembled mnemonics and parameters

prog1.c:

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

Compile using:

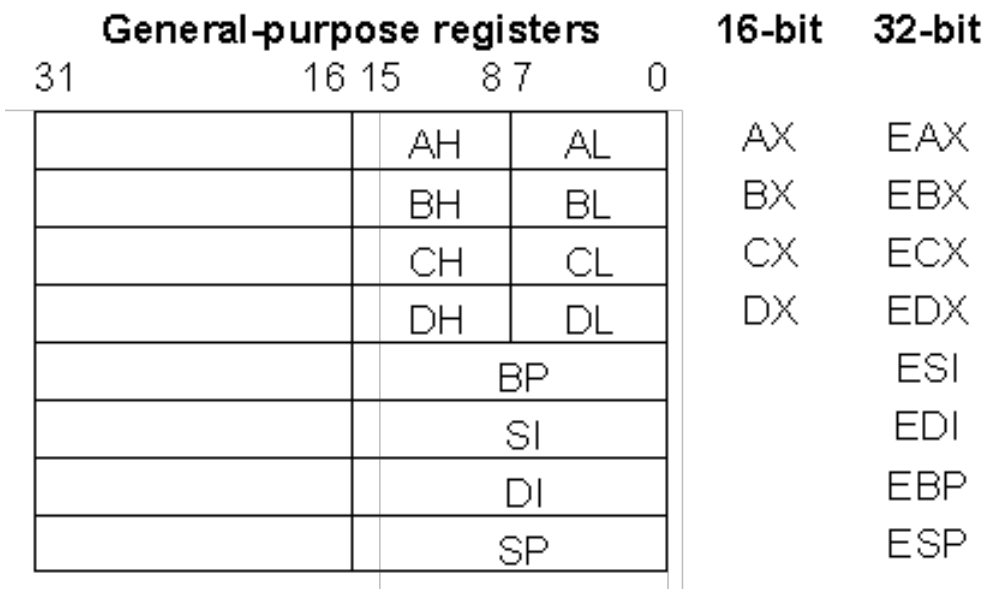
```
gcc -g -o prog1 prog1.c
```

Compiler option "-g" includes additional *debug symbols* in the ELF file

x86 architecture

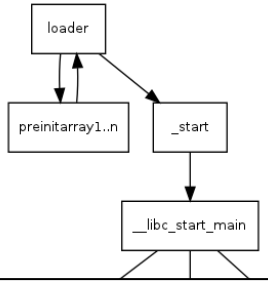
Here: 32 bit architecture („ia32“)

- Registers EAX, EBX, ECX, EDX are 32 bit wide (general purpose*)
  - Also accessible as 16 bit registers AX, BX, CX, DX
  - High/low byte of AX, ..., DX accessible as AH/AL, ..., DH/DL
- Additional registers
  - Stack pointer ESP
  - Base pointer EBP
  - Source (SI) and destination (DI) index
    - e.g. for copy instructions



*almost… not all machine instructions can operate on all four registers

The start() function



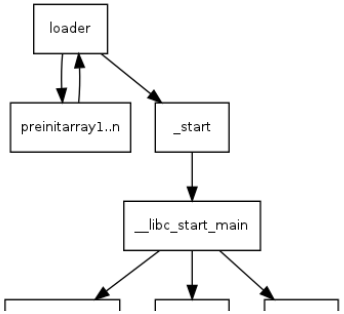

```
080482e0 <_start>:
80482e0: xor    %ebp,%ebp       ; Set register ebp to 0
80482e2: pop    %esi            ; Fetch register esi from the stack
80482e3: mov    %esp,%ecx       ; Copy stack pointer from register ecx
80482e5: and    $0xfffffff0,%esp  ; Stack alignment is multiple of 16 byte
80482e8: push   %eax            ; Save registers eax, esp, edx to stack
80482e9: push   %esp
80482ea: push   %edx
80482eb: push   $0x8048400      ; Save address 0x8048400 on the stack
80482f0: push   $0x80483a0      ; Save address 0x80483a0 on the stack
80482f5: push   %ecx            ; Save registers ecx, esi on the stack
80482f6: push   %esi
80482f7: push   $0x8048394      ; Save address 0x8048394 on the stack
80482fc: call   80482c4 <__libc_start_main@plt> ; call _libc_start_main
8048301: hlt                    ; Stop the processor
```

1 2 3

Three steps:

1. Initialize the stack
2. Save parameters for _libc_start_main on the stack and call it
3. Stop the processor

Passing params to _libc_start_main



Prototype of _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)
);
```

function pointer to our main

argument counter argc and argument string array argv

function pointer to constructor and destructorr for this program*

function pointer for destructor of the dyn. linker (unload shared libraries)

stack pointer

* part of the GNU libc source code in file `csu/libc-start.c`

The C main function

Prototype for main:

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

argc: number of command line arguments (>= 1)

envp: Array containing strings of the shell *environment variables* (null terminated, form A=B)

argv: Array containing strings of the arguments (null terminated strings)
argv[0] = name of the executed program

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

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

Print the environment variables:

```
#include <stdio.h>

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

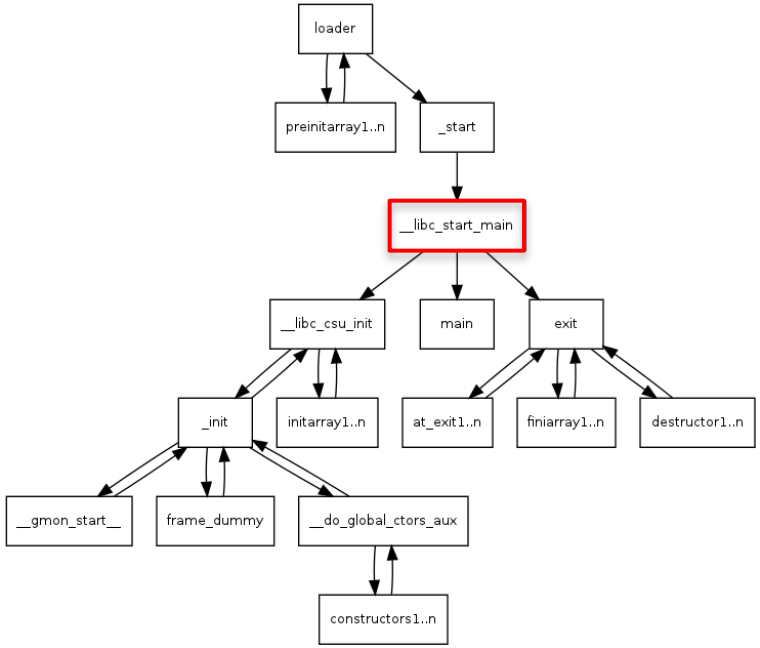
Parameter passing: stack

```
080482e0 <_start>:
80482e0: xor    %ebp,%ebp
80482e2: pop    %esi
80482e3: mov    %esp,%ecx
80482e5: and    $0xfffffff0,%esp
80482e8: push   %eax        ; unused
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 ; address of main
80482fc: call   80482c4 <__libc_start_main@plt>
8048301: hlt
```

Parameters are passed on the stack *in opposite order* of the function signature:

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.

_libc_start_main functionality



Linking and loading

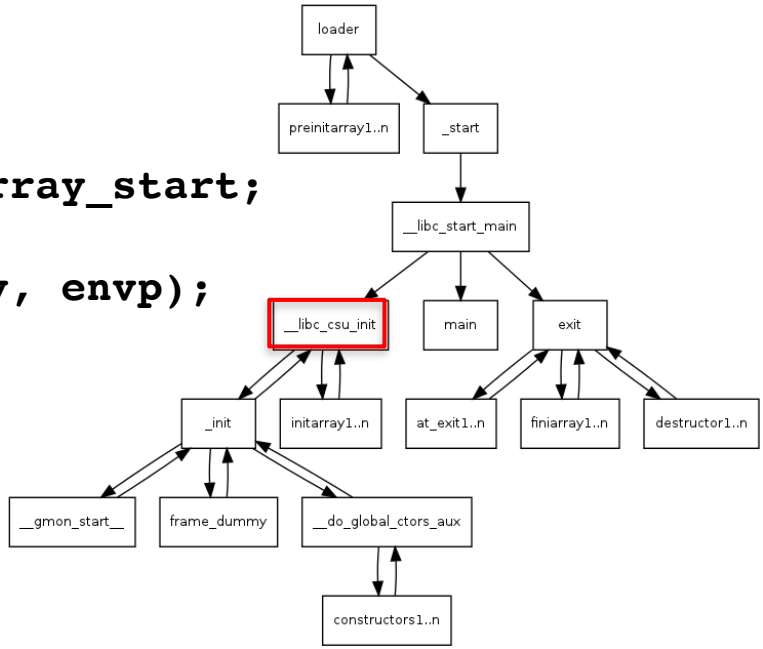
- Takes care of some security problems with setuid setgid programs
- Starts up threading
- Registers the fini (our program), and rtld_fini (run-time loader) arguments to get run by at_exit to run the program's and the loader's cleanup routines
- Calls the init argument
- Calls the main with the argc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

C-Program constructor _libc_csu_init

Wait, this is no C++?!?

- Even C programs have functions to initialise and remove data elements (but not objects)
  - Constructor `__libc_csu_init`, destructor `__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);
}
```



What does _init() do?

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

Output:

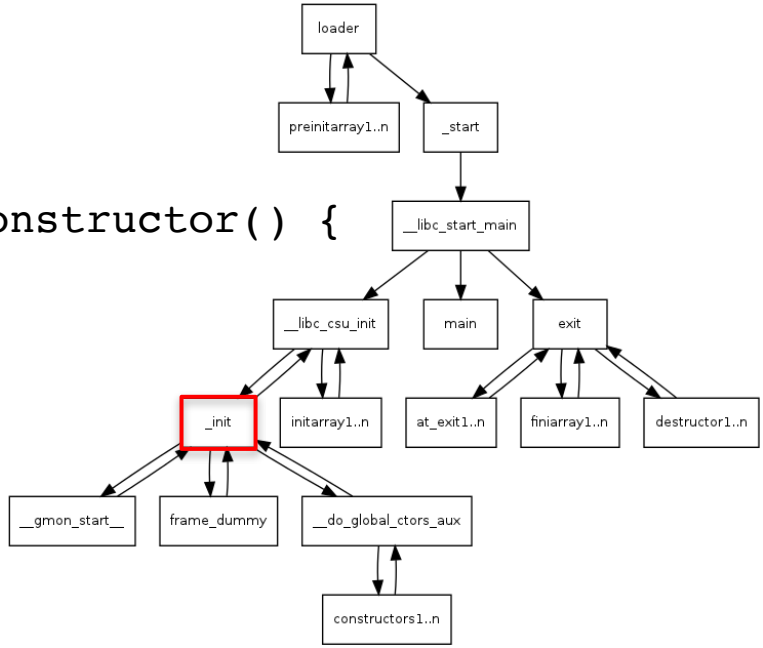
```
$ ./prog2
a_constructor
main
```

Prepare global Information

- Pointer to *global tables*
- Optional start of the *profiler*
- Save information to support *exception handlers*
- Call of *global constructors* (also for C++):

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

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

gcc extension:
provides name of the current function

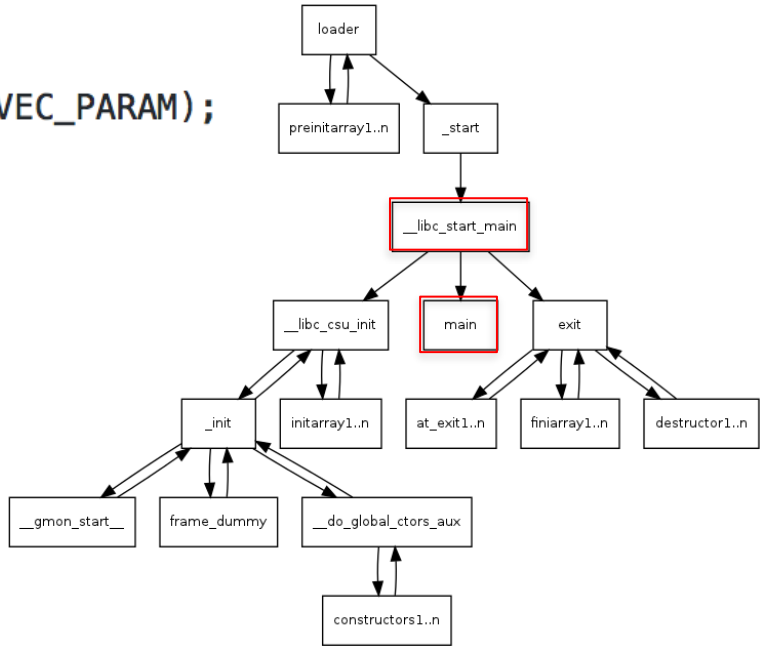


…finally!

Calling our main function is nothing special…

- uses pointer to main function that was passed to id
- passes argc, argv and envp to main on the stack

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



* excerpt from the GNU libc source code in file `csu/libc-start.c`

References

1. ELF TIS Committee, Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification, Version 1.2
   Online unter https://www.uclibc.org/docs/elf.pdf
2. John R. Levine, Linkers and Loaders, MKP 1999, ISBN 1-55860-496-0
   Online (beta) unter https://www.iecc.com/linker/
3. Working with libraries and the linker
   http://www.bottomupcs.com/libraries_and_the_linker.xhtml
4. Ulrich Drepper, How to write shared libraries
   https://software.intel.com/sites/default/files/m/a/1/e/dsohowto.pdf
5. Optimizing linker load times – https://lwn.net/Articles/192624/
6. ABI conventions – https://sites.google.com/site/x32abi/
7. Eric S. Raymond, The Art of UNIX Programming, Addison-Wesley 2003, ISBN-13: 978-0131429017
8. Giuseppe Di Cataldo, Stack Frames – A Look From Inside, Apress 2016, ISBN-13: 978-1-4842-2181-5