Executable Formats, Program Startup, and Binary Manipulation

SAPIENZA
UNIVERSITÀ DI ROMA

Alessandro Pellegrini

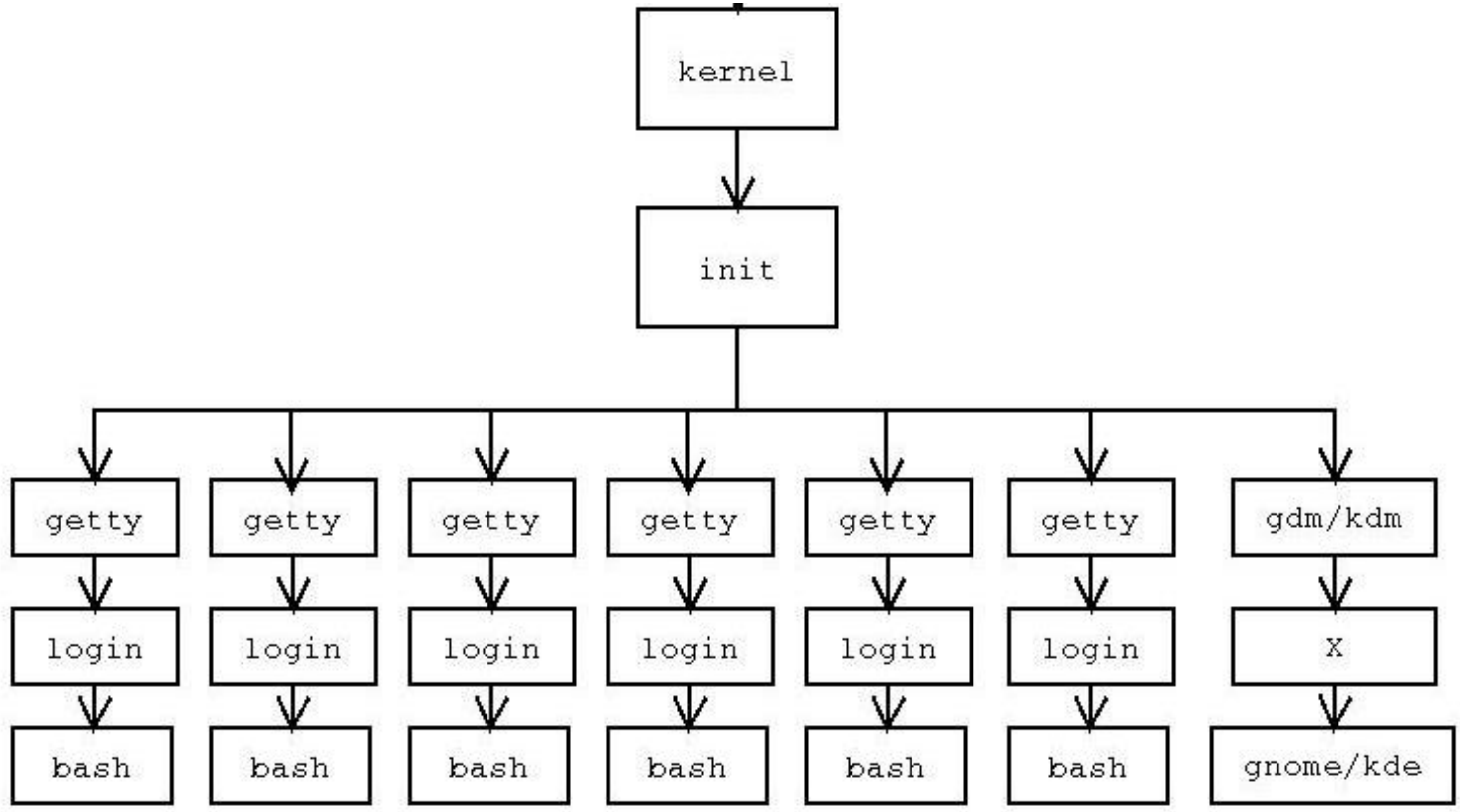
Advanced Operating Systems and Virtualization

How a Program is Started?

- We all know how to compile a program:
  - `gcc  program.c -o program`
- We all know how to launch the compiled program:
  - `./program`

- The question is: why all this works?
- What is the *convention* used between kernel and user space?

In the beginning, there was `init`



Starting a Program from `bash`

```
static int execute_disk_command (char *command, int
pipe_in, int pipe_out, int async, struct fd_bitmap
*fds_to_close) {
  pid_t pid;
  pid = make_child (command, async);

  if (pid == 0) {
    shell_execve (command, args, export_env);
  }
}
```

Starting a Program from bash

```
pid_t make_child (char *command, int async_p) {
  pid_t pid;
  int forksleep;

  start_pipeline();

  forksleep = 1;
  while ((pid = fork ()) < 0 && errno == EAGAIN && forksleep < FORKSLEEP_MAX) {
      sys_error("fork: retry");

      reap_zombie_children();
      if (forksleep > 1 && sleep(forksleep) != 0)
        break;
      forksleep <<= 1;
  }

  if (pid < 0) {
      sys_error ("fork");
      throw_to_top_level ();
  }

  if (pid == 0) {
      sigprocmask (SIG_SETMASK, &top_level_mask, (sigset_t *)NULL);
  } else {
      last_made_pid = pid;
      add_pid (pid, async_p);
  }
  return (pid);
}
```

Starting a Program from bash

```
int shell_execve (char *command, char **args, char **env) {

  execve (command, args, env);

  READ_SAMPLE_BUF (command, sample, sample_len);

  if (sample_len == 0)
    return (EXECUTION_SUCCESS);

  if (sample_len > 0) {
    if (sample_len > 2 && sample[0] == '#' && sample[1] == '!')
      return (execute_shell_script(sample, sample_len, command, args, env));
    else if (check_binary_file (sample, sample_len)) {
      internal_error (_("%s: cannot execute binary file"), command);
      return (EX_BINARY_FILE);
    }
  }

  longjmp(subshell_top_level, 1);
}
```

`fork()` and `exec*()`

- To create a new process, a couple of `fork()` and `exec*()` calls should be issued
  - Unix worked mainly with multiprocessing (shared memory)
  - `fork()` relies on COW
  - `fork()` followed by `exec*()` allows for fast creation of new processes, both for sharing memory view or not

do_fork()

- Fresh PCB/kernel-stack allocation
- Copy/setup of PCB information
- Copy/setup of PCB linked data structures
- What information is copied or inherited (namely shared into the original buffers) depends on the value of the flags passed in input to do_fork()
- Admissible values for the flags are defined in `include/linux/sched.h`
  - `CLONE_VM`: set if VM is shared between processes
  - `CLONE_FS`: set if fs info shared between processes
  - `CLONE_FILES`: set if open files shared between processes
  - `CLONE_PID`: set if pid shared
  - `CLONE_PARENT`: set if we want to have the same parent as the cloner

`exec*()`

- `exec*()` does not create a new process
- it just changes the program file that an existing process is running:
  - It first wipes out the memory state of the calling process
  - It then goes to the filesystem to find the program file requested
  - It copies this file into the program's memory and initializes register state, including the PC
  - It doesn't alter most of the other fields in the PCB
    - the process calling `exec*()` (the child copy of the shell, in this case) can, e.g., change the open files

struct linux_binprm

```
struct linux_binprm {
    char buf[BINPRM_BUF_SIZE];
    struct page *page[MAX_ARG_PAGES];
    unsigned long p; /* current top of mem */
    int sh_bang;
    struct file* file;
    int e_uid, e_gid;
    kernel_cap_t cap_inheritable, cap_permitted, cap_effective;
    int argc, envc;
    char *filename;   /* Name of binary */
    unsigned long loader, exec;
};
```

do_execve()

```
int do_execve(char *filename, char **argv, char **envp, struct pt_regs
*regs) {
    struct linux_binprm bprm;
    struct file *file;
    int retval;
    int i;

    file = open_exec(filename);

    retval = PTR_ERR(file);
    if (IS_ERR(file))
        return retval;

    bprm.p = PAGE_SIZE*MAX_ARG_PAGES-sizeof(void *);
    memset(bprm.page, 0, MAX_ARG_PAGES*sizeof(bprm.page[0]));
    bprm.file = file;
    bprm.filename = filename;
    bprm.sh_bang = 0;
    bprm.loader = 0;
    bprm.exec = 0;

    if ((bprm.argc = count(argv, bprm.p / sizeof(void *))) < 0) {
        allow_write_access(file);
        fput(file);
        return bprm.argc;
    }
```

do_execve()

```
if ((bprm.envc = count(envp, bprm.p / sizeof(void *))) < 0) {
        allow_write_access(file);
        fput(file);
        return bprm.envc;
}

retval = prepare_binprm(&bprm);
if (retval < 0)
        goto out;

retval = copy_strings_kernel(1, &bprm.filename, &bprm);
if (retval < 0)
        goto out;

bprm.exec = bprm.p;
retval = copy_strings(bprm.envc, envp, &bprm);
if (retval < 0)
        goto out;

retval = copy_strings(bprm.argc, argv, &bprm);
if (retval < 0)
        goto out;

retval = search_binary_handler(&bprm,regs);
if (retval >= 0)
        /* execve success */
        return retval;
```

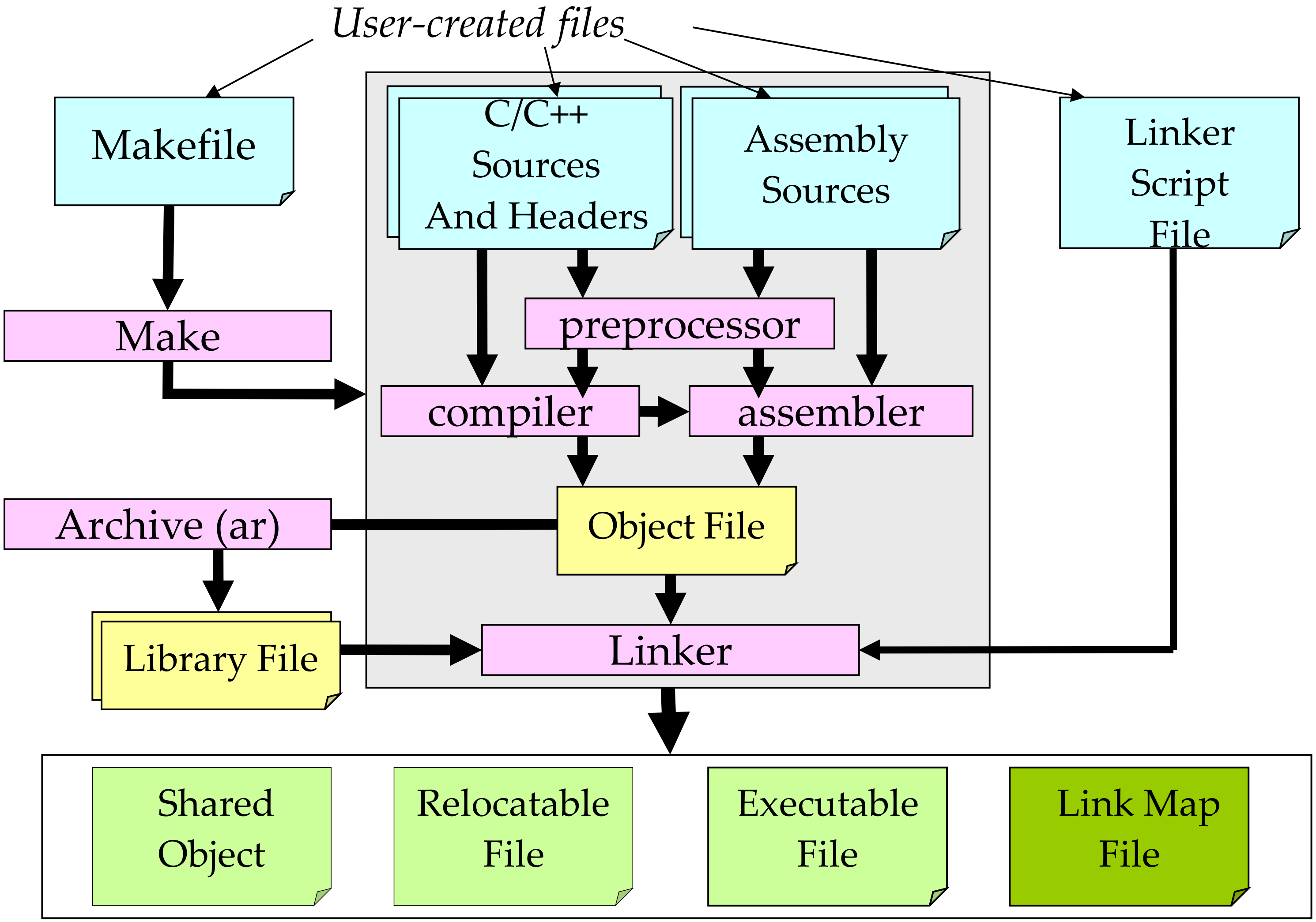
do_execve()

```
  out:
    /* Something went wrong, return the inode and free the argument pages*/
    allow_write_access(bprm.file);
    if (bprm.file)
        fput(bprm.file);

    for (i = 0 ; i < MAX_ARG_PAGES ; i++) {
        struct page * page = bprm.page[i];
        if (page)
            __free_page(page);
    }

    return retval;
}
```

search_binary_handler()

- search_binary_handler():
  - Scans a list of binary file handlers registered in the kernel;
  - If no handler is able to recognize the image format, syscall returs the ENOEXEC error ("Exec Format Error");
- In fs/binfmt_elf.c:
  - load_elf_binary():
    - Load image file to memory using mmap;
    - Reads the program header and sets permissions accordingly
    - **elf_ex = *((struct elfhdr *)bprm->buf);**

Compiling Process



User-created files

Makefile

C/C++ Sources And Headers

Assembly Sources

Linker Script File

Make

preprocessor

compiler

assembler

Object File

Archive (ar)

Library File

Linker

Shared Object

Relocatable File

Executable File

Link Map File

Object File Format

- For more than 20 years, *nix executable file format has been `a.out` (since 1975 to 1998).
- This format was made up of at most 7 sections:
  - *exec header*: loading information;
  - *text segment*: machine instructions;
  - *data segment*: initialized data;
  - *text relocations*: information to update pointers;
  - *data relocations*: information to update pointers;
  - *symbol table*: information on variables and functions;
  - *string table*: names associated with symbols.

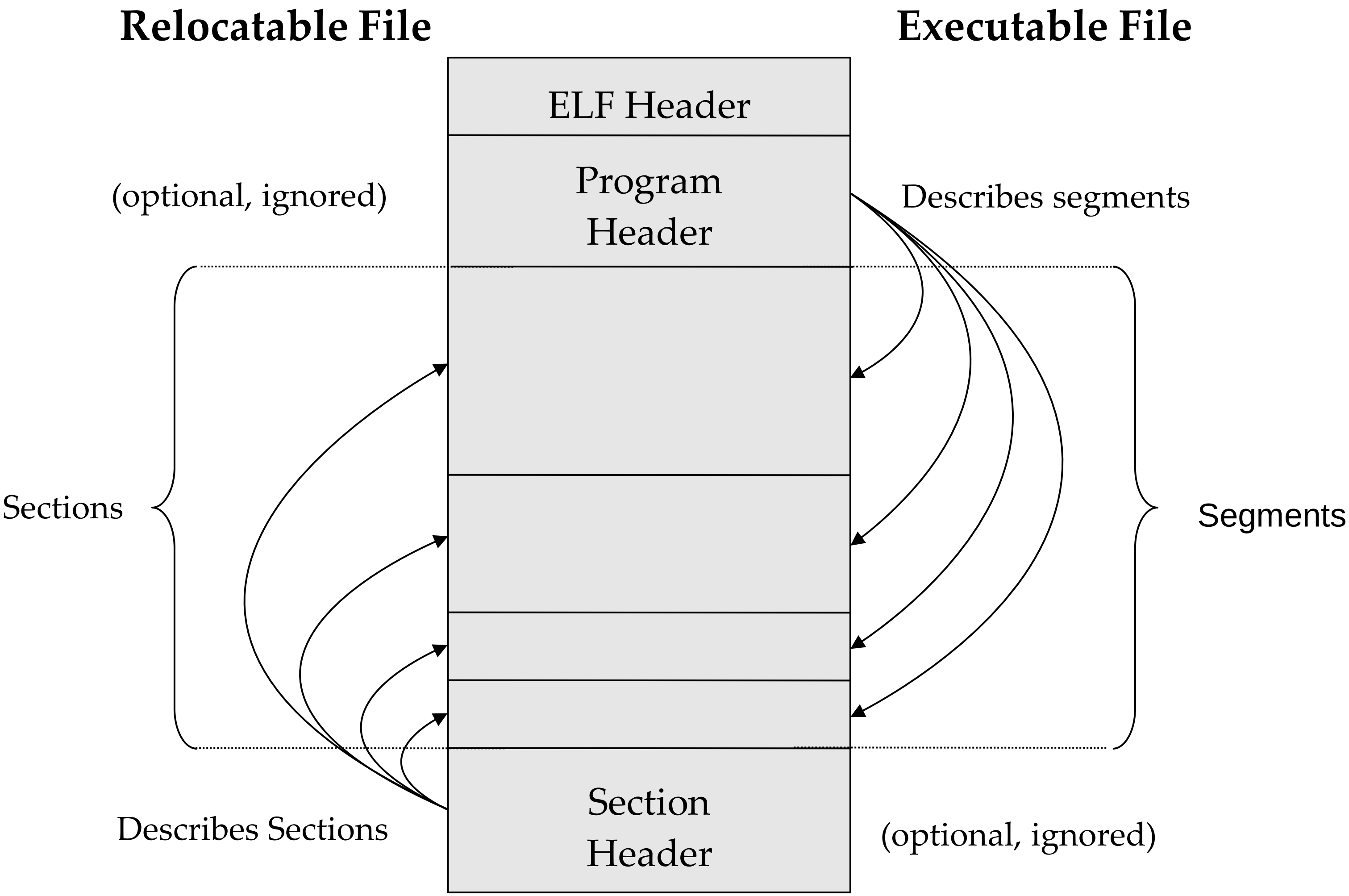
Object File Format

- This format's limits were:
  - cross-compiling;
  - dynamic linking;
  - creation of simple shared libraries;
  - Lack for support of initializers/finalizers (e.g. constructors and destructors).
- Linux has definitively replaced `a.out` with ELF (Executable and Linkable Format) in version 1.2 (more or less in 1995).

ELF Types of Files

- ELF defines the format of binary executables. There are four different categories:
  - *Relocatabale* (Created by compilers and assemblers. Must be processed by the linker before being run).
  - *Executable* (All symbols are resolved, except for shared libraries' symbols, which are resolved at runtime).
  - *Shared object* (A library which is shared by different programs, contains all the symbols' information used by the linker, and the code to be executed at runtime).
  - *Core file* (a core dump).
- ELF files have a twofold nature
  - Compilers, assemblers and linkers handle them as a set of logical sections;
  - The system loader handles them as a set of segments.

ELF File's Structure



ELF Header

```
#define EI_NIDENT (16)

typedef struct {
  unsigned char e_ident[EI_NIDENT];/* Magic number and other info */
  Elf32_Half    e_type;      /* Object file type */
  Elf32_Half    e_machine;   /* Architecture */
  Elf32_Word    e_version;   /* Object file version */
  Elf32_Addr    e_entry;     /* Entry point virtual address */
  Elf32_Off     e_phoff;     /* Program header table file offset */
  Elf32_Off     e_shoff;     /* Section header table file offset */
  Elf32_Word    e_flags;     /* Processor-specific flags */
  Elf32_Half    e_ehsize;    /* ELF header size in bytes */
  Elf32_Half    e_phentsize; /* Program header table entry size */
  Elf32_Half    e_phnum;     /* Program header table entry count */
  Elf32_Half    e_shentsize; /* Section header table entry size */
  Elf32_Half    e_shnum;     /* Section header table entry count */
  Elf32_Half    e_shstrndx;  /* Section header string table index */
} Elf32_Ehdr;
```

Relocatable File

- A **relocatable file** or a **shared object** is a collection of sections
- Each section contains a single kind of information, such as executable code, read-only data, read/write data, relocation entries, or symbols.
- Each symbol's address is defined in relation to the section which contains it.
  - For example, a function's entry point is defined in relation to the section of the program which contains it.

Section Header

```
typedef struct {
  Elf32_Word    sh_name;      /* Section name (string tbl index) */
  Elf32_Word    sh_type;      /* Section type */
  Elf32_Word    sh_flags;     /* Section flags */
  Elf32_Addr    sh_addr;      /* Section virtual addr at execution */
  Elf32_Off     sh_offset;    /* Section file offset */
  Elf32_Word    sh_size;      /* Section size in bytes */
  Elf32_Word    sh_link;      /* Link to another section */
  Elf32_Word    sh_info;      /* Additional section information */
  Elf32_Word    sh_addralign; /* Section alignment */
  Elf32_Word    sh_entsize;   /* Entry size if section holds table */
} Elf32_Shdr;
```

Types and Flags in Section Header

`PROGBITS`: The section contains the program content (code, data, debug information).

`NOBITS`: Same as `PROGBITS`, yet with a null size.

`SYMTAB` and `DYNSYM`: The section contains a symbol table.

`STRTAB`: The section contains a string table.

`REL` and `RELA`: The section contains relocation information.

`DYNAMIC` and `HASH`: The section contains dynamic linking information.

`WRITE`: The section contains runtime-writeable data.

`ALLOC`: The section occupies memory at runtime.

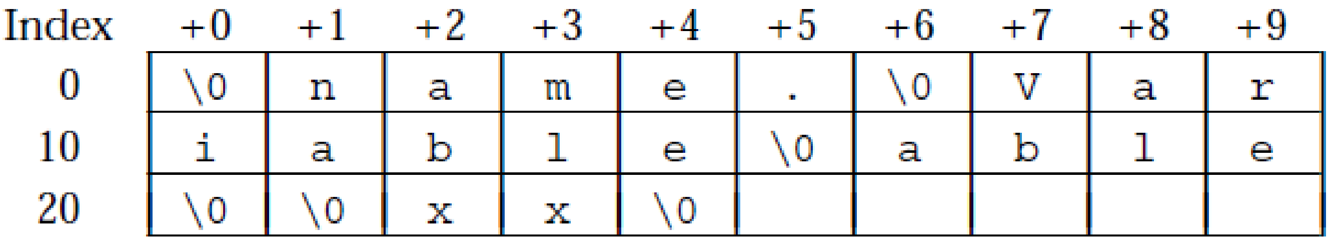
`EXECINSTR`: The section contains executable machine instructions.

Some Sections

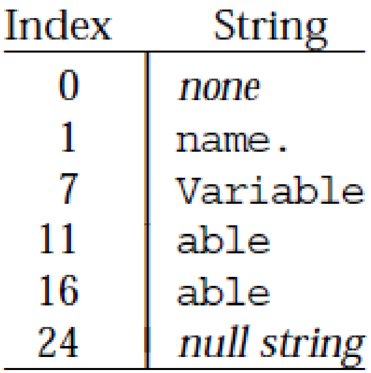
- `.text`: contains program's instructions
  - Type: `PROGBITS`
  - Flags: `ALLOC` + `EXECINSTR`
- `.data`: contains preinitialized read/write data
  - Type: `PROGBITS`
  - Flags: `ALLOC` + `WRITE`
- `.rodata`: contains preinitialized read-only data
  - Type: `PROGBITS`
  - Flags: `ALLOC`
- `.bss`: contains uninitialized data. will be set to zero at startup.
  - Type: `NOBITS`
  - Flags: `ALLOC` + `WRITE`

String Table

- Sections keeping string tables contain sequence of null-terminated strings.
- Object files use a string table to represent symbols' and sections' names.
- A string is referred using an index in the table.
- Symbol table and symbol names are separated because there is no limit in names' length in C/C++



Index	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
0	\0	n	a	m	e	.	\0	V	a	r
10	i	a	b	l	e	\0	a	b	l	e
20	\0	\0	x	x	\0					



Index	String
0	*none*
1	name.
7	Variable
11	able
16	able
24	*null string*

Symbol Table

- The Symbol Table keeps in an object file the information necessary to identify and relocate symbolic definitions in a program and its references.

```
typedef struct {
  Elf32_Word    st_name;   /* Symbol name  */
  Elf32_Addr    st_value;  /* Symbol value */
  Elf32_Word    st_size;   /* Symbol size */
  unsigned char st_info;   /* Symbol binding */
  unsigned char st_other;  /* Symbol visibility */
  Elf32_Section st_shndx;  /* Section index */
} Elf32_Sym;
```

Static Relocation Table

- Relocation is the process which connects references to symbols with definition of symbols.
- Relocatable files must keep information on how to modify the contents of sections.

```
typedef struct {
  Elf32_Addr    r_offset; /* Address */
  Elf32_Word   r_info;   /* Relocation type and symbol index */
} Elf32_Rel;


typedef struct {
  Elf32_Addr    r_offset; /* Address */
  Elf32_Word    r_info;   /* Relocation type and symbol index */
  Elf32_Sword   r_addend; /* Addend */
} Elf32_Rela;
```

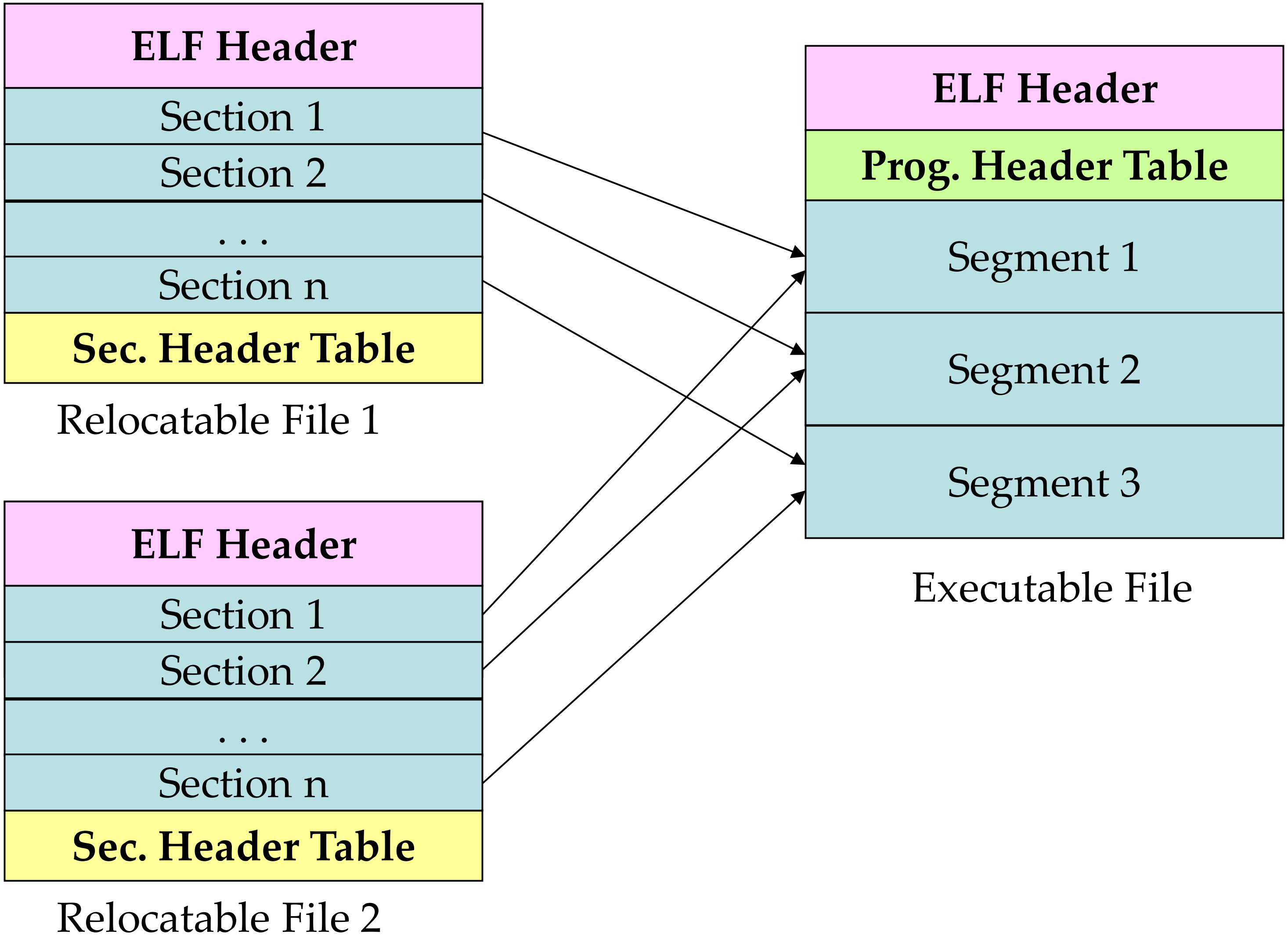
Executable Files

- Usually, an executable file has only few segments:
  - A read-only segment for code.
  - A read-only segment for read-only data.
  - A read/write segment for other data.
- Any section marked with flag `ALLOCATE` is packed in the proper segment, so that the operating system is able to map the file to memory with few operations.
  - If `.data` and `.bss` sections are present, they are placed within the same read/write segment.

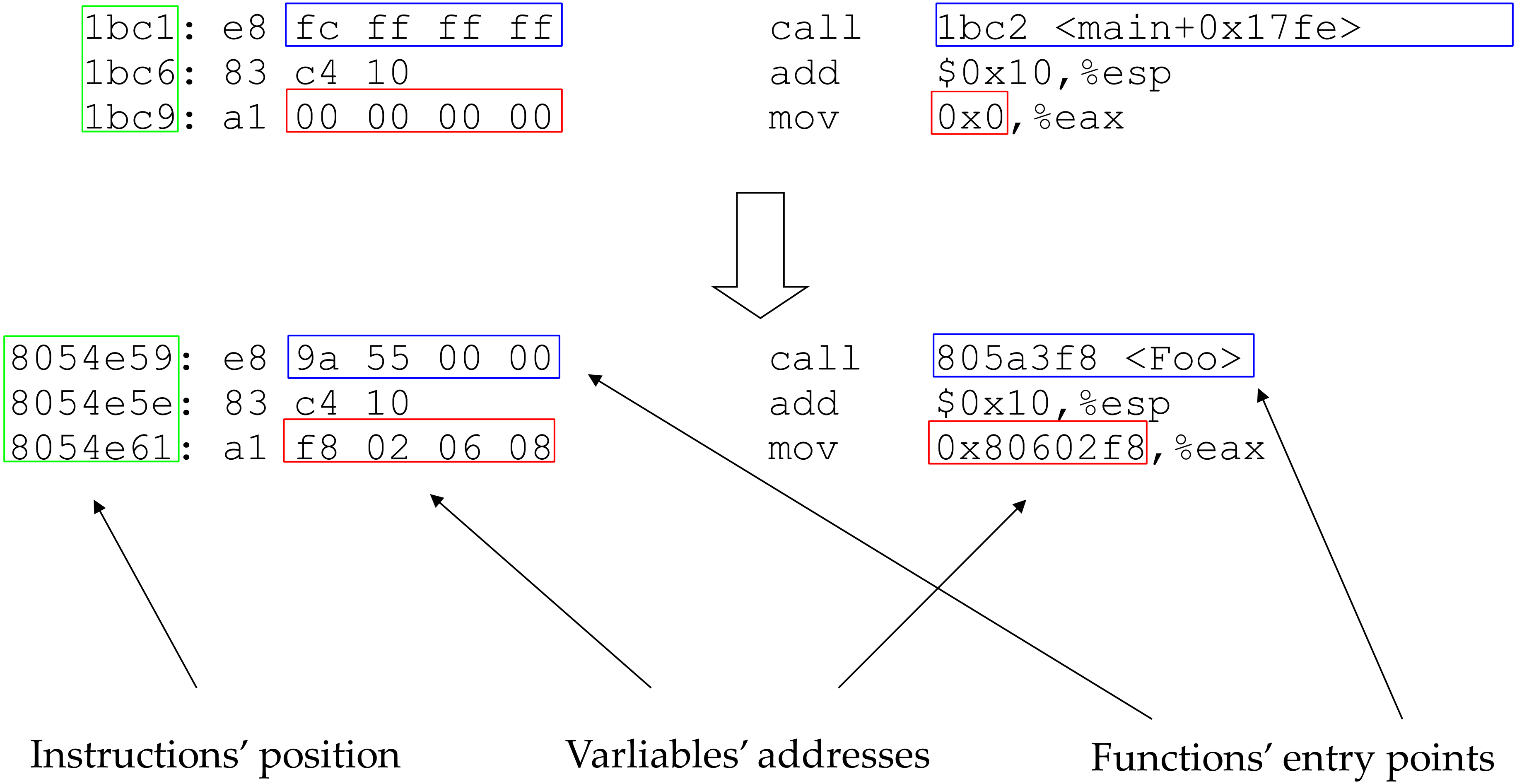
Program Header

```
typedef struct {
  Elf32_Word    p_type;   /* Segment type */
  Elf32_Off     p_offset; /* Segment file offset */
  Elf32_Addr    p_vaddr;  /* Segment virtual address */
  Elf32_Addr    p_paddr;  /* Segment physical address */
  Elf32_Word    p_filesz; /* Segment size in file */
  Elf32_Word    p_memsz;  /* Segment size in memory */
  Elf32_Word    p_flags;  /* Segment flags */
  Elf32_Word    p_align;  /* Segment alignment */
} Elf32_Phdr;
```

Linker's Role



Static Relocation



```
1bc1: e8 fc ff ff ff         call   1bc2 <main+0x17fe>
1bc6: 83 c4 10               add    $0x10,%esp
1bc9: a1 00 00 00 00         mov    0x0,%eax
```

```
8054e59: e8 9a 55 00 00      call   805a3f8 <Foo>
8054e5e: 83 c4 10            add    $0x10,%esp
8054e61: a1 f8 02 06 08      mov    0x80602f8,%eax
```

Instructions’ position

Varliables’ addresses

Functions’ entry points

Directives: Linker Script

- The simplest form of linker script contains only a `SECTIONS` directive;
- The `SECTIONS` directive describes memory layout of the linker-generated file.

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

Sets *location counter*'s value

Places all input files's `.text` sections in the output file's `.text` section at the address specified by the *location counter*.

Example: C code

```c
#include <stdio.h>

int xx, yy;

int main(void) {
  xx = 1;
  yy = 2;
  printf ("xx %d yy %d\n", xx, yy);
}
```

Example: ELF Header

```
$ objdump -x example-program

esempio-elf: file format elf32-i386
architecture: i386,
flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x08048310
```

Example: Program Header

```
   PHDR off    0x00000034 vaddr 0x08048034 paddr 0x08048034 align 2**2
        filesz 0x00000100 memsz 0x00000100 flags r-x
 INTERP off    0x00000134 vaddr 0x08048134 paddr 0x08048134 align 2**0
        filesz 0x00000013 memsz 0x00000013 flags r--
   LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
        filesz 0x000004f4 memsz 0x000004f4 flags r-x
   LOAD off    0x00000f0c vaddr 0x08049f0c paddr 0x08049f0c align 2**12
        filesz 0x00000108 memsz 0x00000118 flags rw-
DYNAMIC off    0x00000f20 vaddr 0x08049f20 paddr 0x08049f20 align 2**2
        filesz 0x000000d0 memsz 0x000000d0 flags rw-
   NOTE off    0x00000148 vaddr 0x08048148 paddr 0x08048148 align 2**2
        filesz 0x00000020 memsz 0x00000020 flags r--
  STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**2
        filesz 0x00000000 memsz 0x00000000 flags rw-
  RELRO off    0x00000f0c vaddr 0x08049f0c paddr 0x08049f0c align 2**0
        filesz 0x000000f4 memsz 0x000000f4 flags r--
```

Example: Dynamic Section

```
NEEDED                    libc.so.6
INIT                      0x08048298
FINI                      0x080484bc
HASH                      0x08048168
STRTAB                    0x08048200
SYMTAB                    0x080481b0
STRSZ                     0x0000004c
SYMENT                    0x00000010
DEBUG                     0x00000000
PLTGOT                    0x08049ff4
PLTRELSZ                  0x00000018
PLTREL                    0x00000011
JMPREL                    0x08048280
```

There is the need to link to this shared library to use printf()

Example: Section Header

```
Idx Name             Size      VMA       LMA       File off  Algn

2   .hash            00000028  08048168  08048168  00000168  2**2
                     CONTENTS, ALLOC, LOAD, READONLY, DATA
10 .init             00000030  08048298  08048298  00000298  2**2
                     CONTENTS, ALLOC, LOAD, READONLY, CODE
11 .plt              00000040  080482c8  080482c8  000002c8  2**2
                     CONTENTS, ALLOC, LOAD, READONLY, CODE
12 .text             000001ac  08048310  08048310  00000310  2**4
                     CONTENTS, ALLOC, LOAD, READONLY, CODE
13 .fini             0000001c  080484bc  080484bc  000004bc  2**2
                     CONTENTS, ALLOC, LOAD, READONLY, CODE
14 .rodata           00000015  080484d8  080484d8  000004d8  2**2
                     CONTENTS, ALLOC, LOAD, READONLY, ATA
22 .data             00000008  0804a00c  0804a00c  0000100c  2**2
                     CONTENTS, ALLOC, LOAD, DATA
23 .bss              00000010  0804a014  0804a014  00001014  2**2
                     ALLOC
```

Example: Symbol Table

```
...
00000000 l    df *ABS*     00000000          esempio-elf.c
08049f0c l       .ctors    00000000          .hidden __init_array_end
08049f0c l       .ctors    00000000          .hidden __init_array_start
08049f20 l     O .dynamic  00000000          .hidden _DYNAMIC
0804a00c  w      .data     00000000          data_start
08048420 g     F .text     00000005          __libc_csu_fini
08048310 g     F .text     00000000          _start
00000000  w      *UND*     00000000          __gmon_start__
...
08049f18 g     O .dtors    00000000          .hidden __DTOR_END__
08048430 g     F .text     0000005a          __libc_csu_init
00000000       F *UND*     00000000          printf@@GLIBC_2.0
0804a01c g     O .bss      00000004          yy
0804a014 g       *ABS*     00000000          __bss_start
0804a024 g       *ABS*     00000000          _end
0804a014 g       *ABS*     00000000          _edata
0804848a g     F .text     00000000         .hidden __i686.get_pc_thunk.bx
080483c4 g     F .text     0000004d          main
08048298 g     F .init     00000000          _init
0804a020 g     O .bss      00000004          xx
```

Symbols Visibility

- *weak* symbols:
  - More modules can have a symbol with the same name of a weak one;
  - The declared entity cannot be overloaded by other modules;
  - It is useful for libraries which want to avoid conflicts with user programs.
- gcc version 4.0 gives the command line option `-fvisibility`:
  - *default*: normal behaviour, the symbol is seen by other modules;
  - *hidden*: two declarations of an object refer the same object only if they are in the same shared object;
  - *internal*: an entity declared in a module cannot be referenced even by pointer;
  - *protected*: the symbol is weak;

Symbols Visibility

```
int variable __attribute__ ((visibility (“hidden”)));
```

```
#pragma GCC visibility push(hidden)
int variable;

int increment(void) {
    return ++variable;
}
#pragma GCC visibility pop
```

Entry Point for the Program

- `main()` is not the actual entry point for the program
- glibc inserts auxiliary functions
  - The actual entry point is called `_start`
- The Kernel starts the *dynamic linker* which is stored in the `.interp` section of the program (usually `/lib/ld-linux.so.2`)
- If no dynamic linker is specified, control is given at address specified in `e_entry`

Dynamic Linker

- Initialization steps:
  - Self initialization
  - Loading Shared Libraries
  - Resolving remaining relocations
  - Transfer control to the application
- The most important data structures which are filled are:
  - Procedure Linkage Table (PLT), used to call functions whose address isn't known at link time
  - Global Offsets Table (GOT), similarly used to resolve addresses of data/functions

Dynamic Relocation Data Structures

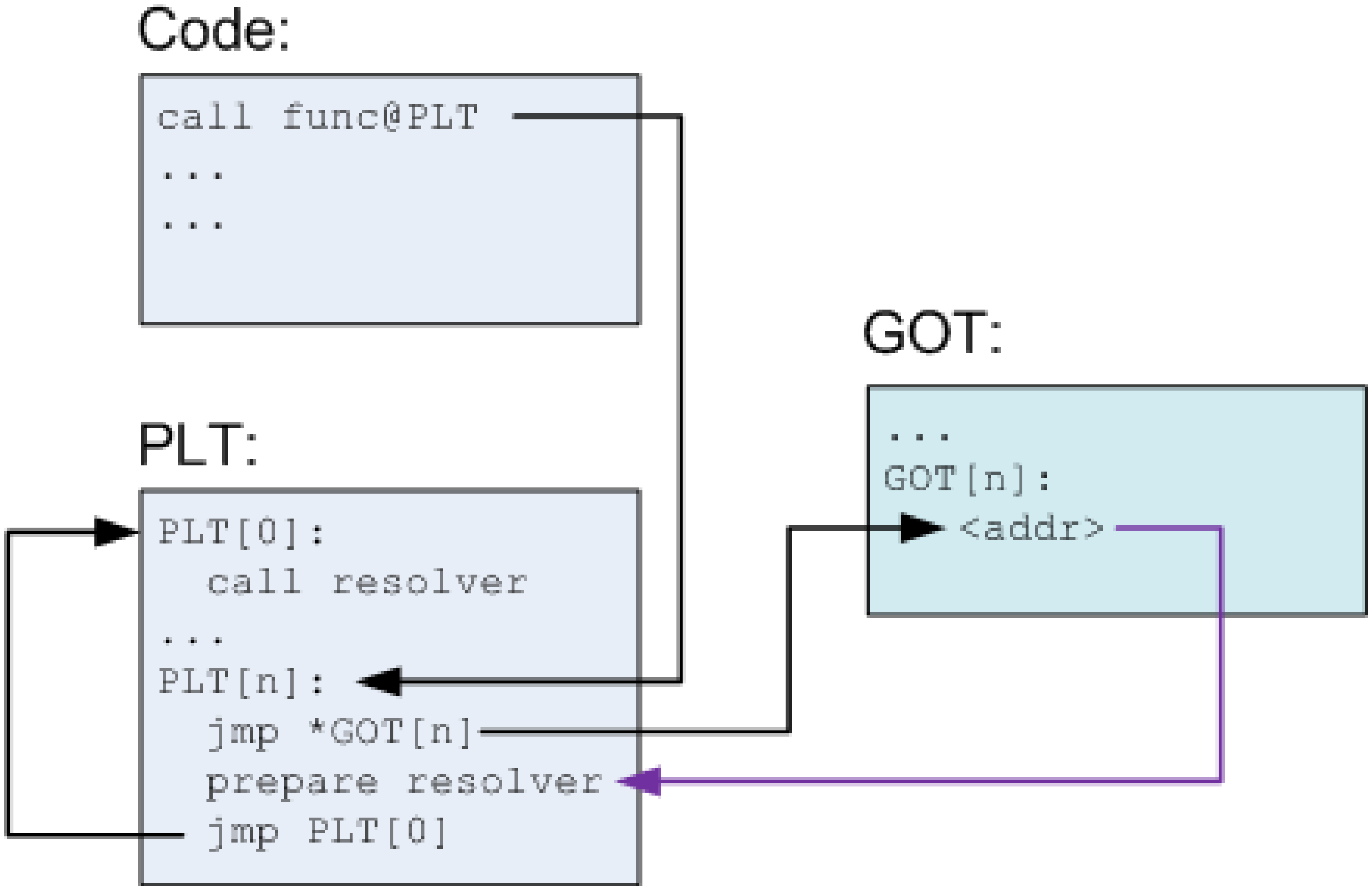
- `.dynsym`: a minimal symbol table used by the dynamic linker when performing relocations
- `.hash`: a hash table that is used to quickly locate a given symbol in the `.dynsym`, usually in one or two tries.
- `.dynstr`: string table related to the symbols stored in `.dynsym`

- This tables are used to populate the GOT table
- This table is populate upon need (*lazy binding*)

Steps to populate the tables

- The PLT first entry is special
- Other entries are identical, one for each function needing resolution.
  - A jump to a location which is specified in a corresponding GOT entry
  - Preparation of arguments for a *resolver* routine
  - Call to the resolver routine, which resides in the first entry of the PLT
- The first PLT entry is a call to the *resolver* located in the dynamic loader itself

GOT and PLT after library loading



Code:

```
call  func@PLT
...
...
```

PLT:

```
PLT[0]:
   call  resolver
...
PLT[n]:
   jmp  *GOT[n]
   prepare  resolver
   jmp  PLT[0]
```

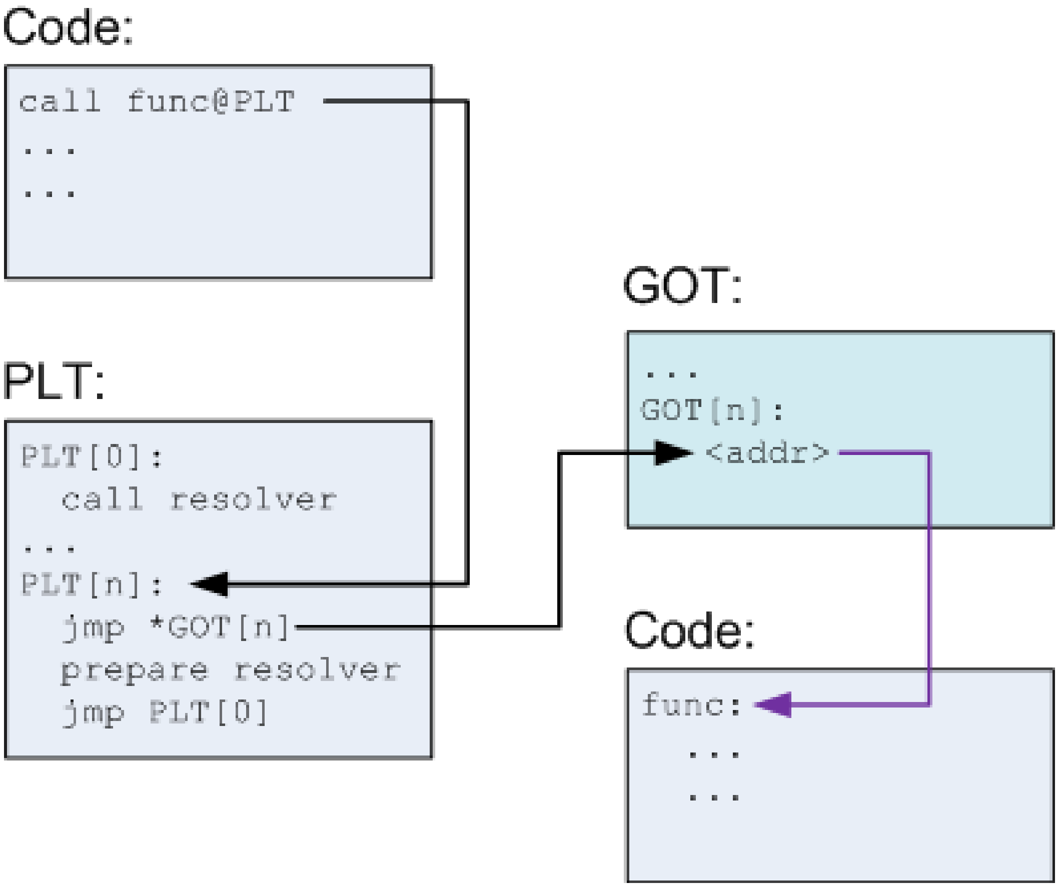
GOT:

```
...
GOT[n]:
    <addr>
```

Steps to populate the tables

- When `func` is called for the first time:
  - `PLT[n]` is called, and jumps to the address pointed to it in `GOT[n]`
  - This address points into `PLT[n]` itself, to the preparation of arguments for the resolver.
  - The resolver is then called, by jumping to `PLT[0]`
  - The resolver performs resolution of the actual address of `func`, places its actual address into `GOT[n]` and calls `func`.

GOT and PLT after first call to `func`



Initial steps of the Program's Life

- So far the dynamic linker has loaded the shared libraries in memory
- GOT is populated when the program requires certain functions
- Then, the dynamic linker calls `_start`

```
<_start>:
  xor     %ebp,%ebp
  pop     %esi
  mov     %esp,%ecx
  and     $0xfffffff0,%esp
  push    %eax
  push    %esp
  push    %edx
  push    $0x8048600
  push    $0x8048670
  push    %ecx
  push    %esi
  push    $0x804841c
  call    8048338 <__libc_start_main>
  hlt
  nop
  nop
```

Suggested by ABI to mark outermost frame

the pop makes `argc` go into `%esi`

%esp is now pointing at argv. The mov puts argv into %ecx without moving the stack pointer

Align the stack pointer to a multiple of 16 bytes

Prepare parameters to __libc_start_main
%eax is garbage, to keep the alignment

This instruction should be never executed!

__libc_start_main()

- This function is defined as:

```
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

);
```

- `__start()` pushes parameters in reverse order on stack

Explanation of Parameters

__libc_start_main arg	content
Don't know.	Don't care.
void (*stack_end)	Our aligned stack pointer.
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.
void (*fini)(void)	__libc_csu_fini - Destructor of this program. Registered by __libc_start_main with __cxat_exit().
void (*init)(void)	__libc_csu_init, Constructor of this program. Called by __libc_start_main before main.
char **ubp_av	argv off of the stack.
arcg	argc off of the stack.
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.

…what about environment variables?

- There are no environment variables passed here!
- `__libc_start_main` calls `__libc_init_first`
  - It finds the first argument after the `NULL` terminating `argv`
  - Sets the global variable `__environ`
- __libc_start_main uses the same trick
  - After the `NULL` terminating `envp` there is another vector
  - This is the **ELF Auxiliary table**
  - It holds information used by the loader

ELF Auxiliary Table

- Setting the environment variable `LD_SHOW_AUXV=1` before running the program dumps its content

```
$ LD_SHOW_AUXV=1 ./example-program
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 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: ./example-program
AT_PLATFORM: i686
```

`__libc_start_main()`

- 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 `__libc_csu_init` which calls `__init`
- 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

_init()

- This is the *program's constructor*
  - Constructors came far before C++!
- Three main steps:
  - If `gmon_start` in the PLT is not null, the program is being profiled. So `gmon_start` is called to setup profiling
  - Call `frame_dummy`, which sets up parameters to calls `__register_frame_info`: this sets up frame unwinding for exceptions management
  - Last call is done to invoke recursively actual constructors: `_do_global_ctors_aux`

_do_global_ctors_aux()

- This is defined in gcc's source code in crtstuff.c

```
__do_global_ctors_aux (void) {
  func_ptr *p;
  for (p = __CTOR_END__ - 1; *p != (func_ptr) -1; p--)
  (*p) ();
}
```

- __CTOR_END__ is a global variable keeping the number of constructors available for the program

How to implement a Constructor

- It's gcc stuff, so we can use a gcc attribute

```
#include <stdio.h>

void __attribute__ ((constructor)) a_constructor() {
    printf("%s\n", __FUNCTION__);
}
```

- `a_constructor()` will be called right before giving control to `main()`

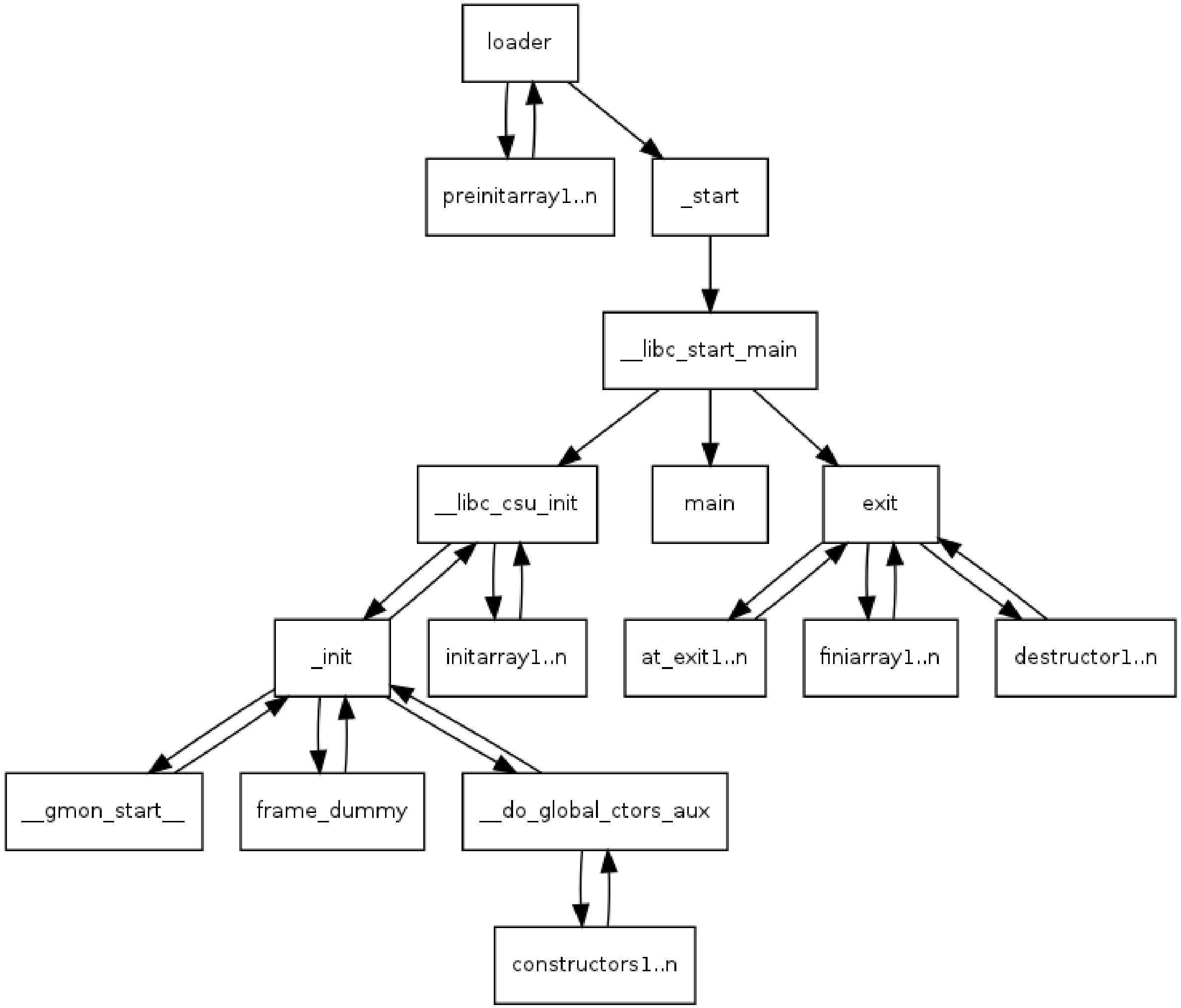
Back to `__libc_csu_init()`

```
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);
}
```

- Again, we can directly run code here, getting arguments as well
- We can hook a function pointer in this way:

```
__attribute__((section(".init_array")))
typeof(init_function) *__init = init_function;
```

The Final Picture



Using this all together

```
#include <stdio.h>

void preinit(int argc, char **argv, char **envp) {
  printf("%s\n", __FUNCTION__);
}
void init(int argc, char **argv, char **envp) {
 printf("%s\n", __FUNCTION__);
}
void fini() {
 printf("%s\n", __FUNCTION__);
}
__attribute__((section(".init_array"))) typeof(init)
*__init = init;
__attribute__((section(".preinit_array"))) typeof(preinit)
*__preinit = preinit;
__attribute__((section(".fini_array"))) typeof(fini)
*__fini = fini;
```

Using this all together

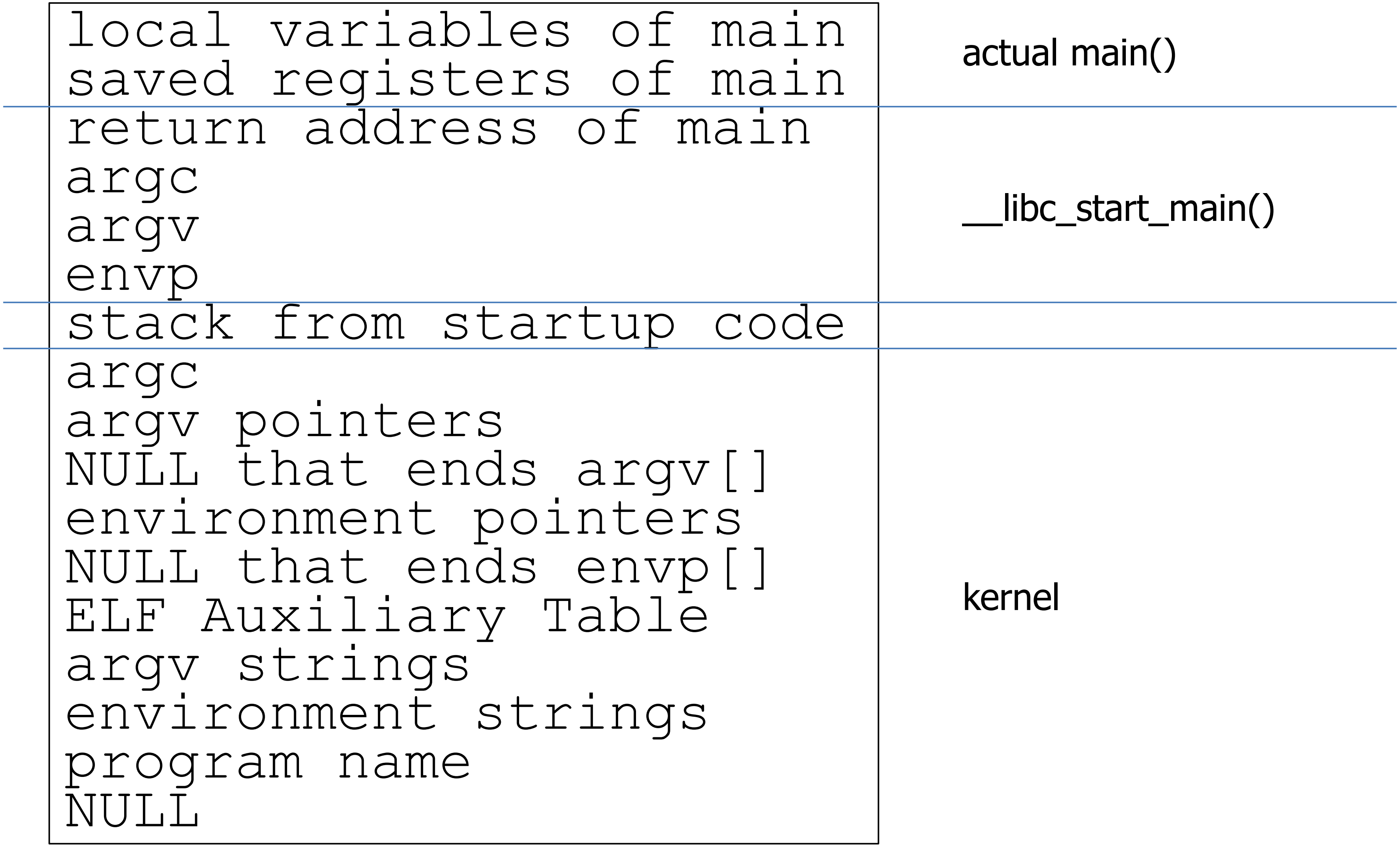
```
void  __attribute__ ((constructor)) constructor() {
 printf("%s\n", __FUNCTION__);
}

void __attribute__ ((destructor)) destructor() {
 printf("%s\n", __FUNCTION__);
}

void my_atexit() {
 printf("%s\n", __FUNCTION__);
}

void my_atexit2() {
 printf("%s\n", __FUNCTION__);
}

int main() {
 atexit(my_atexit);
 atexit(my_atexit2);
}
```

Using this all together

- Compiling and running this program gives this output:

```
$ ./hooks
preinit
constructor
init
my_atexit2
my_atexit
fini
destructor
```

Stack Layout at Program Startup



local variables of main
saved registers of main

actual main()

return address of main
argc
argv
envp

__libc_start_main()

stack from startup code

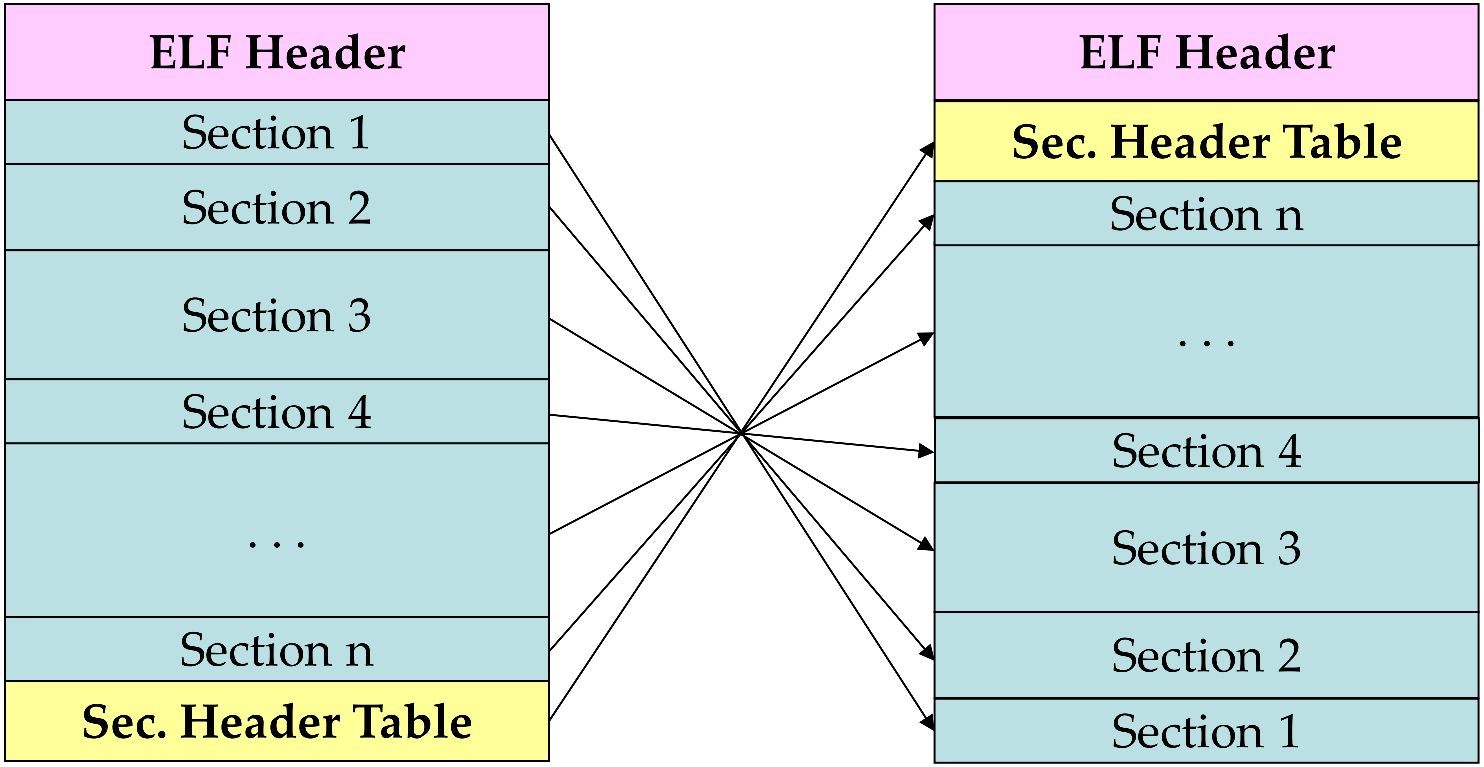
argc
argv pointers
NULL that ends argv[]
environment pointers
NULL that ends envp[]
ELF Auxiliary Table
argv strings
environment strings
program name
NULL

kernel

Manipulating Executables: Code Instrumentation

- Write a userspace program which modifies an ELF, keeping consistent the compilation/loading chain
- Problems:
  - Must work at machine-code level
  - it is important to keep *references coherence* in the code;
  - It is necessary to interpret the original program's code, to find the *right positions* in the code where to inject instrumentation code.
- Used in in *debugging* and in *vulnerability assessment*.

Manipulating ELF: Reordering



Manipulating ELF: Reordering

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <elf.h>
```

To access structures describing and ELF file

```
int main(int argc, char **argv) {

        int elf_src, elf_dst, file_size, i;
        char *src_image, *dst_image, *ptr;
        Elf32_Ehdr *ehdr_src, *ehdr_dst;
        Elf32_Shdr *shdr_src, *shdr_dst;

        if((elf_src = open(argv[1], O_RDONLY)) == -1) exit(-1);
        if((elf_dst = creat(argv[2], 0644)) == -1) exit(-1);
        file_size = lseek(elf_src, 0L, SEEK_END);
        lseek(elf_src, 0L, SEEK_SET);
        src_image = malloc(file_size);
        ptr = dst_image = malloc(file_size);
        read(elf_src, src_image, file_size);
        ehdr_src = (Elf32_Ehdr *)src_image;
        ehdr_dst = (Elf32_Ehdr *)dst_image;

        memcpy(ptr, src_image, sizeof(Elf32_Ehdr));
        ptr += sizeof(Elf32_Ehdr);
```

The two ELF header are (mostly) the same

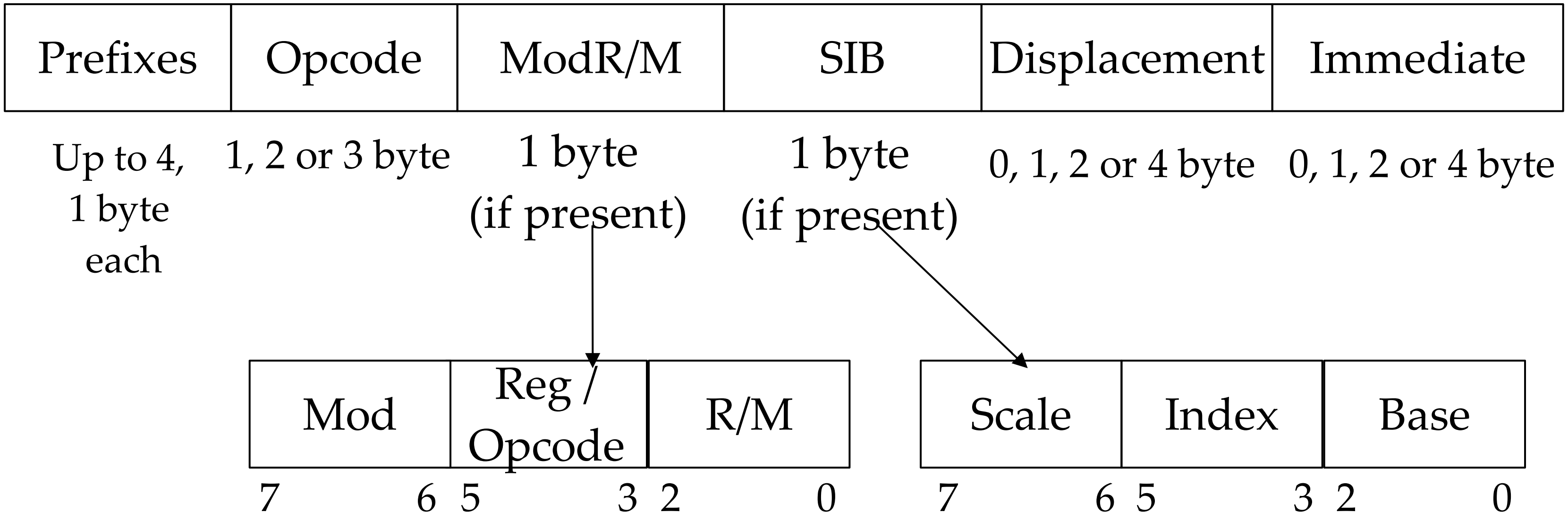
Manipulating ELF: Reordering

```
shdr_dst = (Elf32_Shdr *)ptr;
shdr_src = (Elf32_Shdr *)(src_image + ehdr_src->e_shoff);
ehdr_dst->e_shoff = sizeof(Elf32_Ehdr);
ptr += ehdr_src->e_shnum * ehdr_dst->e_shentsize;

memcpy(shdr_dst, shdr_src, sizeof(Elf32_Shdr));

for(i = ehdr_src->e_shnum - 1; i > 0; i--) {

     memcpy(shdr_dst + ehdr_src->e_shnum - i, shdr_src + i, sizeof(Elf32_Shdr));
     memcpy(ptr, src_image + shdr_src[i].sh_offset, shdr_src[i].sh_size);
     shdr_dst[ehdr_src->e_shnum - i].sh_offset = ptr - dst_image;

    if(shdr_src[i].sh_link > 0)
        shdr_dst[ehdr_src->e_shnum - i].sh_link = ehdr_src->e_shnum - shdr_src[i].sh_link;

    if(shdr_src[i].sh_info > 0)
         shdr_dst[ehdr_src->e_shnum - i].sh_info = ehdr_src->e_shnum - shdr_src[i].sh_info;
    ptr += shdr_src[i].sh_size;
}

ehdr_dst->e_shstrndx = ehdr_src->e_shnum - ehdr_src->e_shstrndx;

write(elf_dst, dst_image, file_size);
close(elf_src);
close(elf_dst);
```

Corrects the header position in the file

Copies sections and headers

Instruction Set: x86



Prefixes	Opcode	ModR/M	SIB	Displacement	Immediate
Up to 4, 1 byte each	1, 2 or 3 byte	1 byte (if present)	1 byte (if present)	0, 1, 2 or 4 byte	0, 1, 2 or 4 byte

Mod	Reg / Opcode	R/M
7 6	5 3	2 0

Scale	Index	Base
7 6	5 3	2 0

Instructions are therefore of variable length (with an upper bound of 15 bytes):

```
85 c0                      test    %eax,%eax
75 09                      jnz     4c
c7 45 ec 00 00 00 00       movl    $0x0,-0x14(%ebp)
eb 59                      jmp     a5
8b 45 08                   mov     0x8(%ebp),%eax
8d 4c 24 04                lea     0x4(%esp),%ecx
0f b7 40 2e                movzwl  0x2e(%eax),%eax
```

Opcode,
ModR/M,
SIB,
Displacement,
Immediate

x86 Addressing Mode

$$\left\{\begin{matrix} CS: \\ DS: \\ SS: \\ ES: \\ FS: \\ GS: \end{matrix}\right\} \left[\left\{\begin{matrix} EAX \\ EBX \\ ECX \\ EDX \\ ESP \\ EBP \\ ESI \\ EDI \end{matrix}\right\}\right] + \left[\left\{\begin{matrix} EAX \\ EBX \\ ECX \\ EDX \\ EBP \\ ESI \\ EDI \end{matrix}\right\} * \left\{\begin{matrix} 1 \\ 2 \\ 4 \\ 8 \end{matrix}\right\}\right] + [displacement]$$

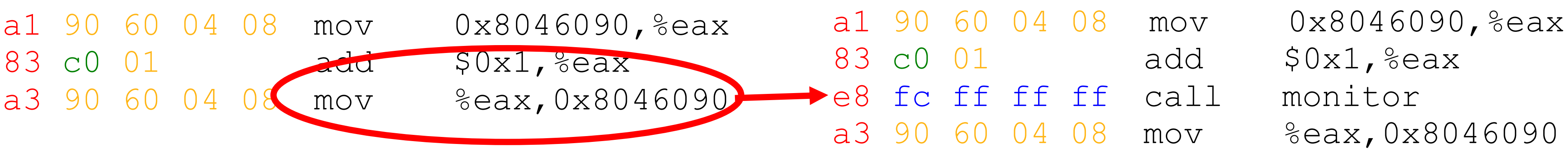
- R/M fields in ModR/M byte and Scale /Index fields in SIB byte identify registers;
- General purpose registers are numbered from da 0 a 7 in this order: `eax` (000), `ecx` (001), `edx` (010), `ebx` (011), `esp` (100), `ebp` (101), `esi` (110), `edi` (111).

Tracking Memory Updates

- Section Header Table is scanned looking for sections containing code (type: `PROGBITS`, flag: `EXECINSTR`);
- Each section is parsed one byte by one;
- Using an opcode-family table the instructions are disassembled, identifying the instructions which have as destination operand a memory location (global variables or dynamically allocated memory);
- Destination operand is decomposed in *base*, *index*, *scale* and *offset*.

Monitor Hooking

- A monitoring routine is hooked by injecting before any memory-write instruction a call to a routine called `monitor`;



```
a1 90 60 04 08    mov      0x8046090,%eax
83 c0 01          add      $0x1,%eax
a3 90 60 04 08    mov      %eax,0x8046090
```

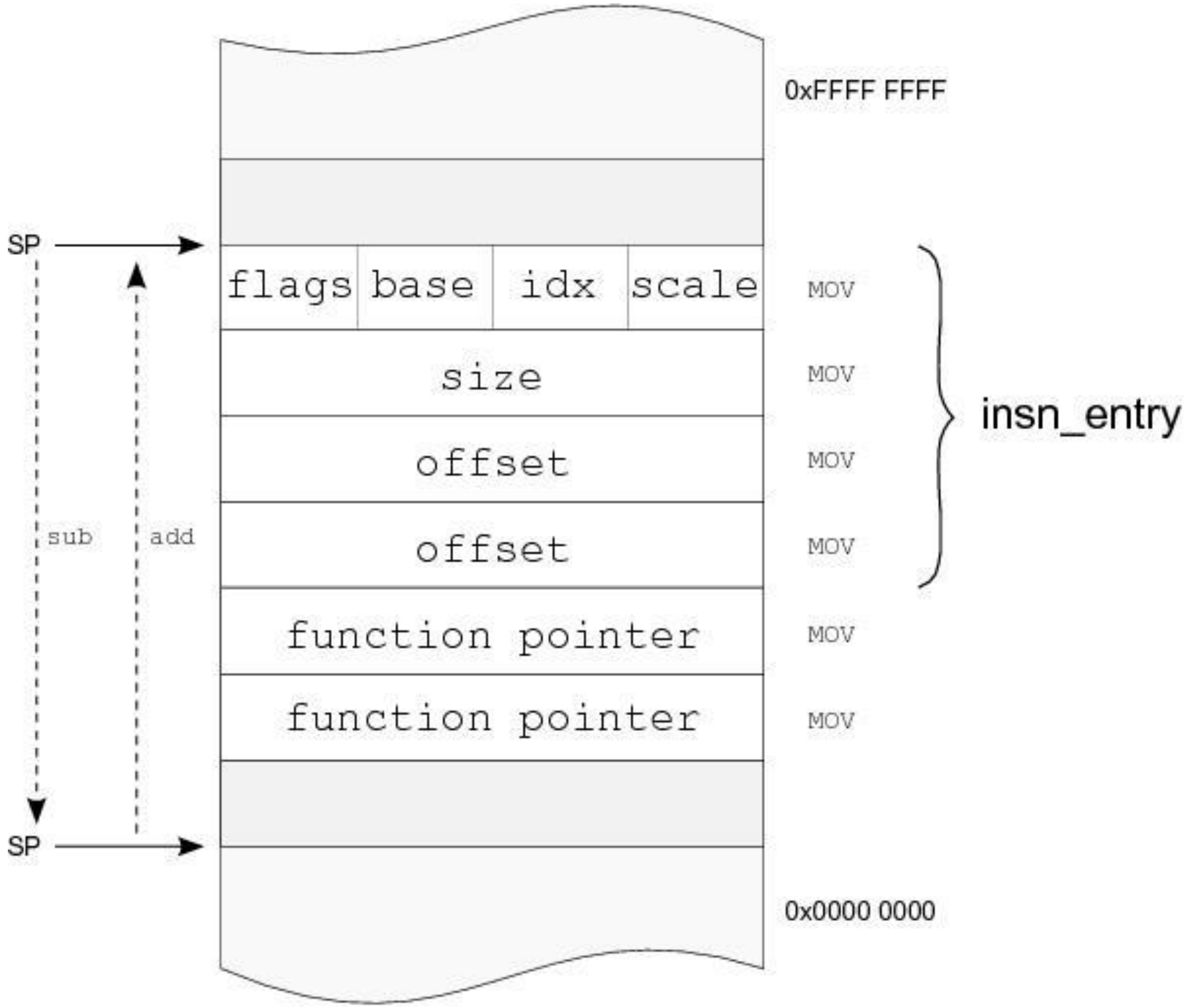
```
a1 90 60 04 08    mov      0x8046090,%eax
83 c0 01          add      $0x1,%eax
e8 fc ff ff ff    call     monitor
a3 90 60 04 08    mov      %eax,0x8046090
```

- We use a call instead of a less costly jump because, by relying on th ereturn value, it is possible to know which original instruction caused the invocation of the monitor;
- Due to this calls insertion, the original sections must be resized (using previously-seen techniques) and relocation tables must be corrected.

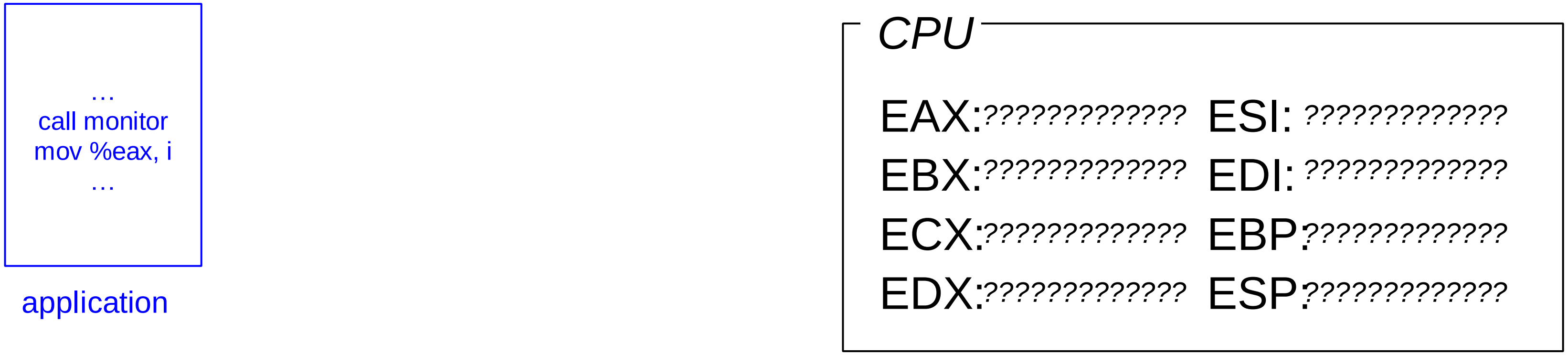
References Correction

- Due to the insertion of instructions, references between portions of code/data are now inconsistent;
- We must therefore:
  - Correct functions entry points;
  - Correct every branch instruction
- Intra-segment jumps in i386 are expressed as offsets starting from the current value of `eip` register, when executing the instruction;
- To correct them, it is necessary to scan the program text a second time and apply a correction to this offset, depending on the amount of bytes inserted in the code;

Caching Dissassembly Information



Memory Trace Execution



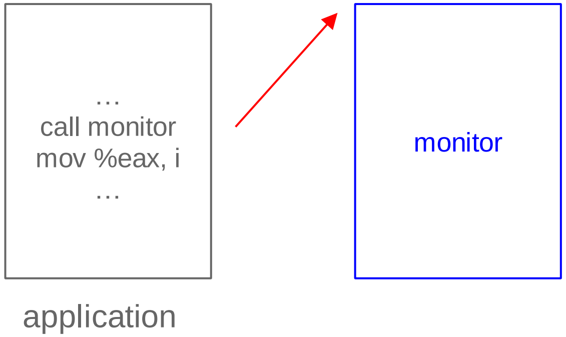
…
call monitor
mov %eax, i
…

application

CPU

EAX:*?????????????* ESI: *?????????????*
EBX:*?????????????* EDI: *?????????????*
ECX:*?????????????* EBP:*?????????????*
EDX:*?????????????* ESP:*?????????????*

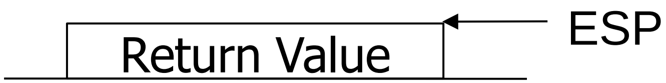
Memory Trace Execution



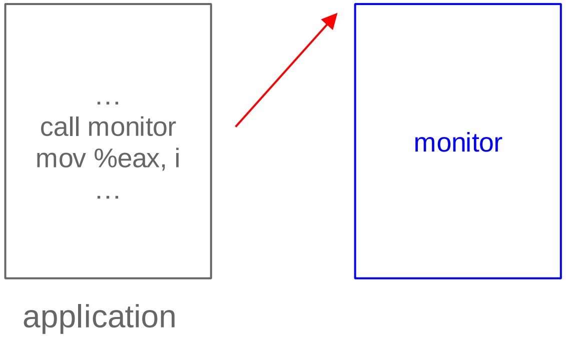
CPU

EAX:????????????? ESI: ?????????????
EBX:????????????? EDI: ?????????????
ECX:????????????? EBP:?????????????
EDX:????????????? ESP:?????????????

```
monitor:
        push    %eax
        push    %ecx
        push    %edx
        push    %ebx
        mov     %esp, %eax
        sub     $4, %esp
        add     $16, %eax
        mov     %eax, (%esp)
        push    %ebp
        push    %esi
        push    %edi
        pushfw
         mov     14(%esp), %ebp
        sub     $4, %ebp
```



Memory Trace Execution



…
call monitor
mov %eax, i
…

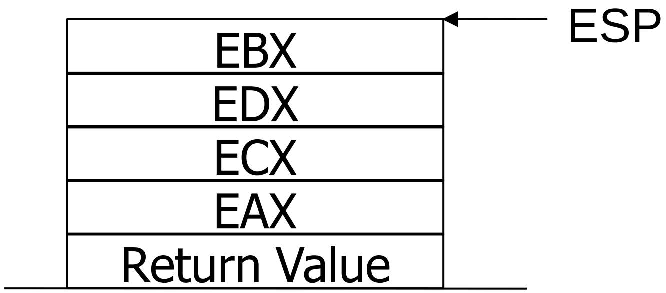
application

monitor

CPU

EAX:*?????????????* ESI: *?????????????*
EBX:*?????????????* EDI: *?????????????*
ECX:*?????????????* EBP:*?????????????*
EDX:*?????????????* ESP:*?????????????*

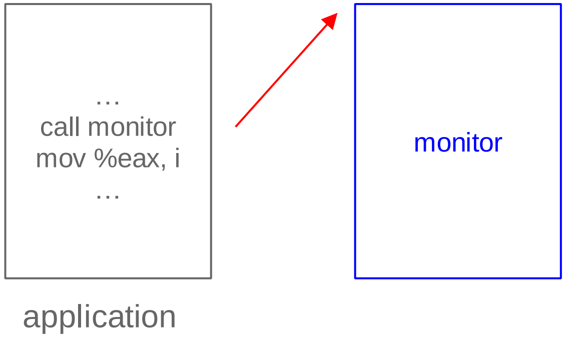
```
monitor:
        push    %eax
        push    %ecx
        push    %edx
        push    %ebx
        mov     %esp, %eax
        sub     $4, %esp
        add     $16, %eax
        mov     %eax, (%esp)
        push    %ebp
        push    %esi
        push    %edi
        pushfw
         mov     14(%esp), %ebp
        sub     $4, %ebp
```



ESP

EBX
EDX
ECX
EAX
Return Value

Memory Trace Execution



…
call monitor
mov %eax, i
…

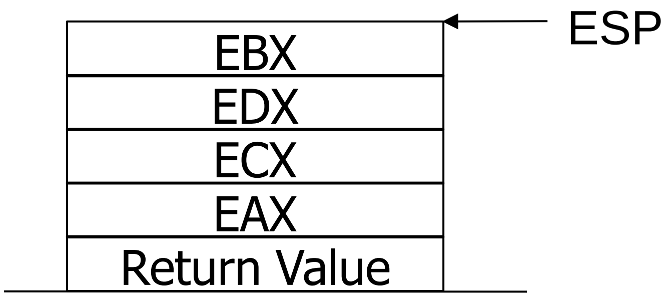
monitor

application

CPU

EAX:*current esp* ESI: ?????????????
EBX:????????????? EDI: ?????????????
ECX:????????????? EBP:?????????????
EDX:????????????? ESP:?????????????

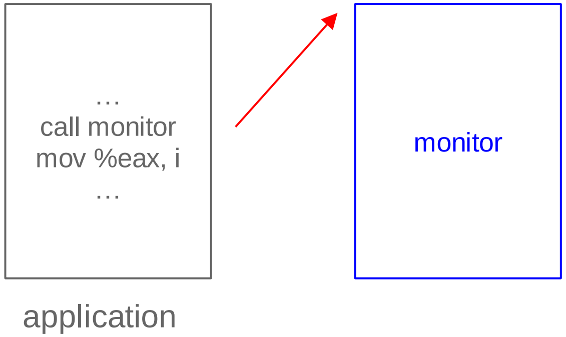
```
monitor:
        push    %eax
        push    %ecx
        push    %edx
        push    %ebx
        mov     %esp, %eax
        sub     $4, %esp
        add     $16, %eax
        mov     %eax, (%esp)
        push    %ebp
        push    %esi
        push    %edi
        pushfw
         mov     14(%esp), %ebp
        sub     $4, %ebp
```



ESP

EBX
EDX
ECX
EAX
Return Value

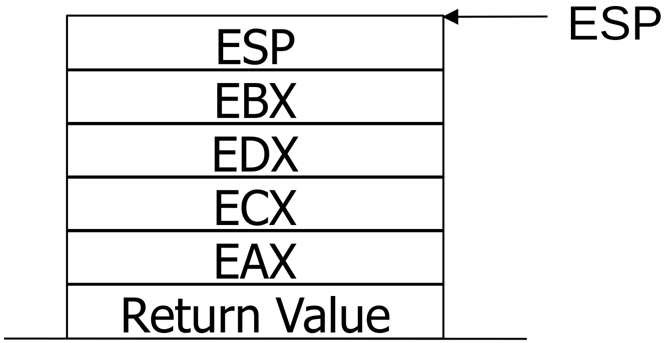
Memory Trace Execution



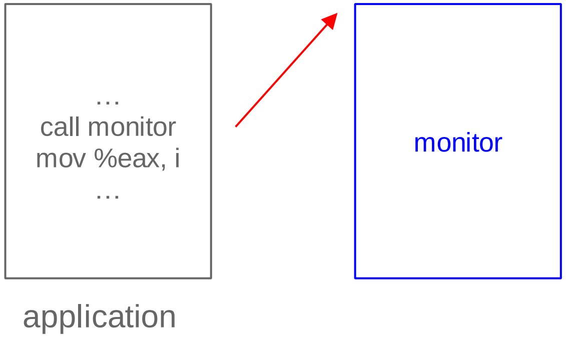
CPU

EAX:*original esp* ESI: *?????????????*
EBX:*?????????????* EDI: *?????????????*
ECX:*?????????????* EBP:*?????????????*
EDX:*?????????????* ESP:*?????????????*

```
monitor:
          push     %eax
          push     %ecx
          push     %edx
          push     %ebx
          mov      %esp, %eax
          sub      $4, %esp
          add      $16, %eax
          mov      %eax, (%esp)
          push     %ebp
          push     %esi
          push     %edi
          pushfw
           mov      14(%esp), %ebp
          sub      $4, %ebp
```



Memory Trace Execution



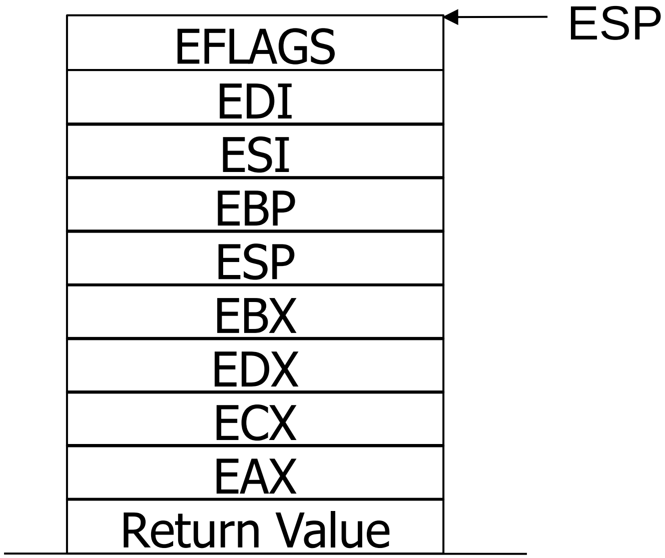
…
call monitor
mov %eax, i
…

application

monitor

CPU

EAX:*original esp* ESI: *?????????????*
EBX:*?????????????* EDI: *?????????????*
ECX:*?????????????* EBP:*?????????????*
EDX:*?????????????* ESP:*?????????????*

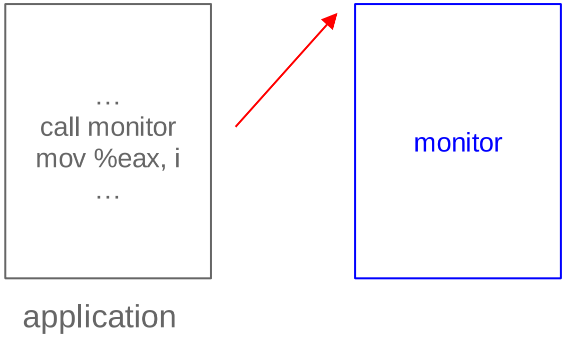


ESP

EFLAGS
EDI
ESI
EBP
ESP
EBX
EDX
ECX
EAX
Return Value

```
monitor:
        push    %eax
        push    %ecx
        push    %edx
        push    %ebx
        mov     %esp, %eax
        sub     $4, %esp
        add     $16, %eax
        mov     %eax, (%esp)
        push    %ebp
        push    %esi
        push    %edi
        pushfw
         mov     14(%esp), %ebp
        sub     $4, %ebp
```

Memory Trace Execution



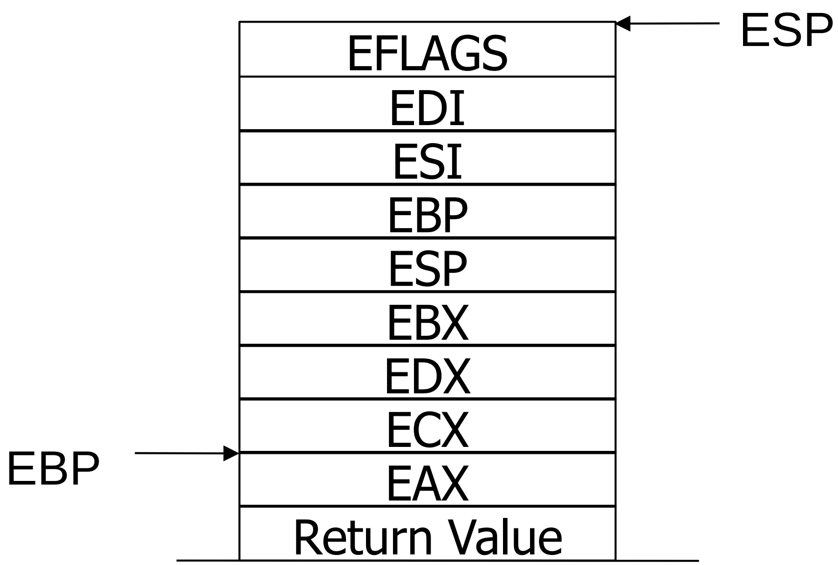
…
call monitor
mov %eax, i
…

application

monitor

CPU

EAX: *original esp* ESI: *?????????????*
EBX: *?????????????* EDI: *?????????????*
ECX: *?????????????* EBP: *orig. eax addr.*
EDX: *?????????????* ESP: *?????????????*



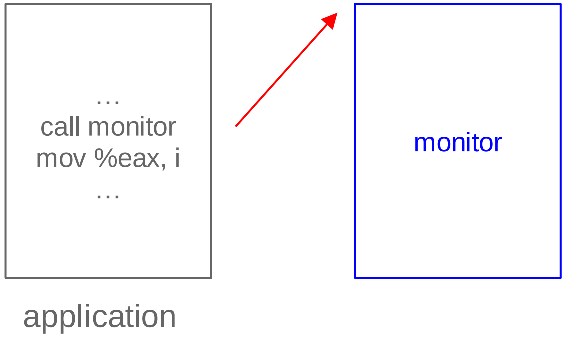
ESP

EFLAGS
EDI
ESI
EBP
ESP
EBX
EDX
ECX
EAX
Return Value

EBP

```
monitor:
        push    %eax
        push    %ecx
        push    %edx
        push    %ebx
        mov     %esp, %eax
        sub     $4, %esp
        add     $16, %eax
        mov     %eax, (%esp)
        push    %ebp
        push    %esi
        push    %edi
        pushfw
        mov     14(%esp), %ebp
        sub     $4, %ebp
```

Memory Trace Execution


…
call monitor
mov %eax, i
…

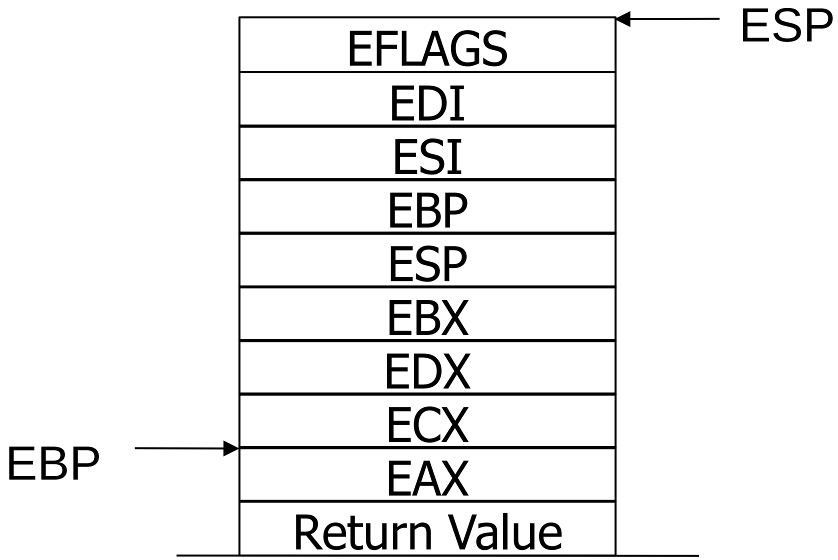
monitor

application

CPU

EAX:	*flags*	ESI:	*?????????????*
EBX:	*???????????????*	EDI:	*?????????????*
ECX:	*??????????????*	EBP:	*indirizzo eax orig.*
EDX:	*flags ptr*	ESP:	*?????????????*

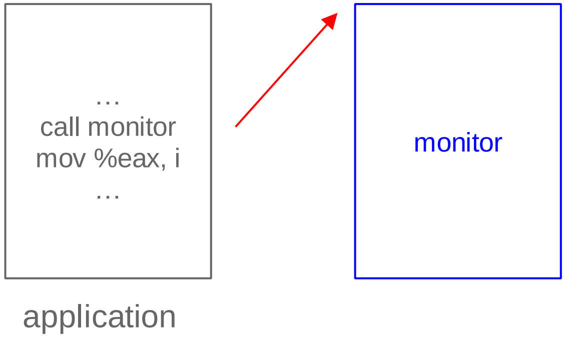
```
monitor:
           lea 16(%ebp), %edx
           movsbl   4(%edx), %eax
           xor      %edi, %edi
          testb    $4, %al
          jz       .NoIndex
           movsbq   6(%edx), %ecx
          negl     %ecx
          movl     (%ebp, %ecx, 4), %edi
          movsbq   7(%edx), %ecx
          imul     %ecx, %edi
```


ESP

EFLAGS
EDI
ESI
EBP
ESP
EBX
EDX
ECX
EAX
Return Value

EBP

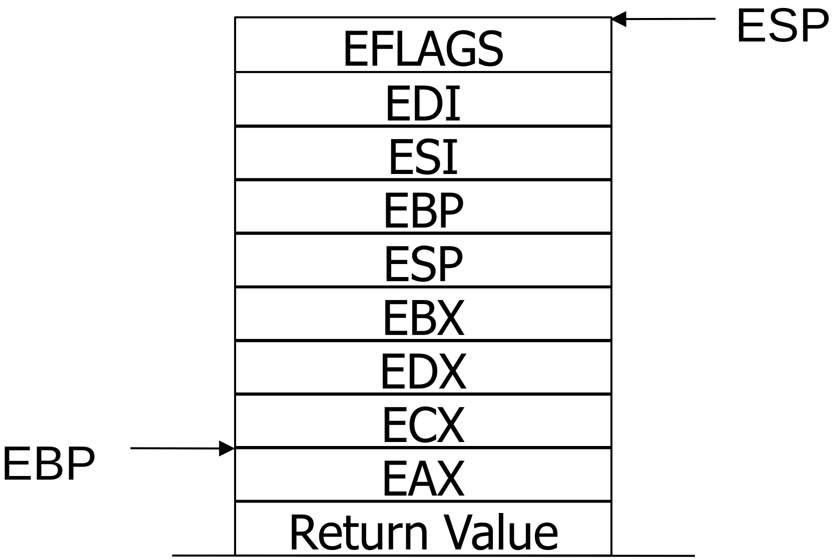
Memory Trace Execution



CPU

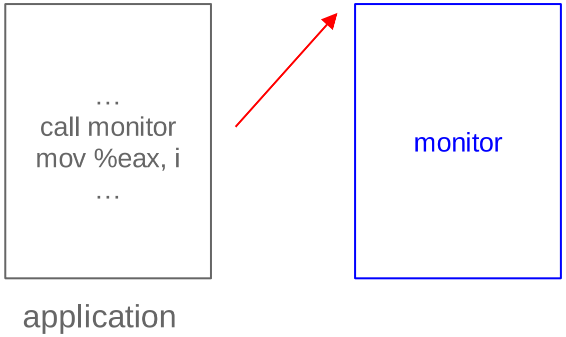
EAX: flags
EBX: ???????????????
ECX: - Idx register
EDX: flags ptr
ESI: ?????????????
EDI: idx
EBP: indirizzo eax orig.
ESP: ?????????????

```
monitor:
           lea 16(%ebp), %edx
           movsbl   4(%edx), %eax
           xor       %edi, %edi
          testb    $4, %al
          jz       .NoIndex
           movsbq   6(%edx), %ecx
          negl     %ecx
          movl     (%ebp, %ecx, 4), %edi
          movsbq   7(%edx), %ecx
          imul     %ecx, %edi
```



EFLAGS
EDI
ESI
EBP
ESP
EBX
EDX
ECX
EAX
Return Value

Memory Trace Execution



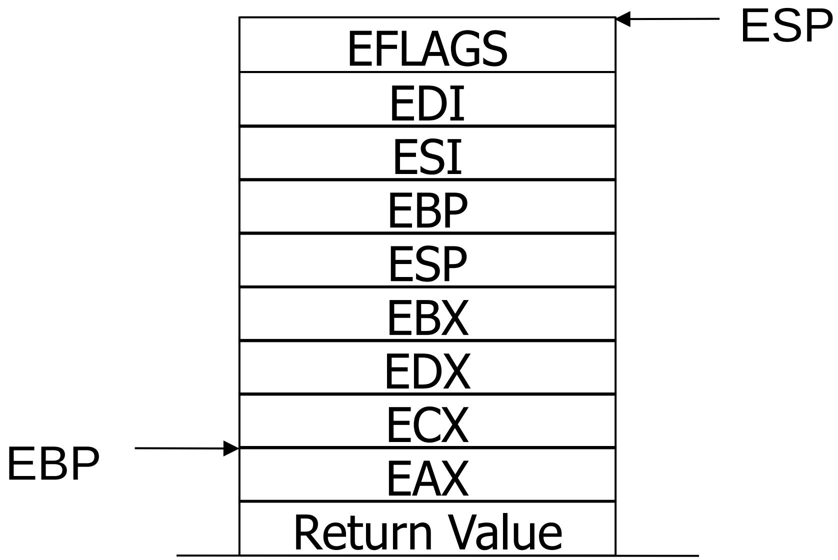
…
call monitor
mov %eax, i
…

application

monitor

CPU

EAX: flags
EBX: ???????????????
ECX: scale
EDX: flags ptr
ESI: ?????????????
EDI: idx * scale
EBP: indirizzo eax orig.
ESP: ?????????????



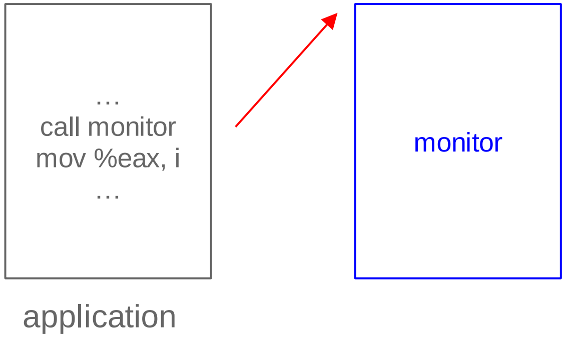
ESP

EFLAGS
EDI
ESI
EBP
ESP
EBX
EDX
ECX
EAX
Return Value

EBP

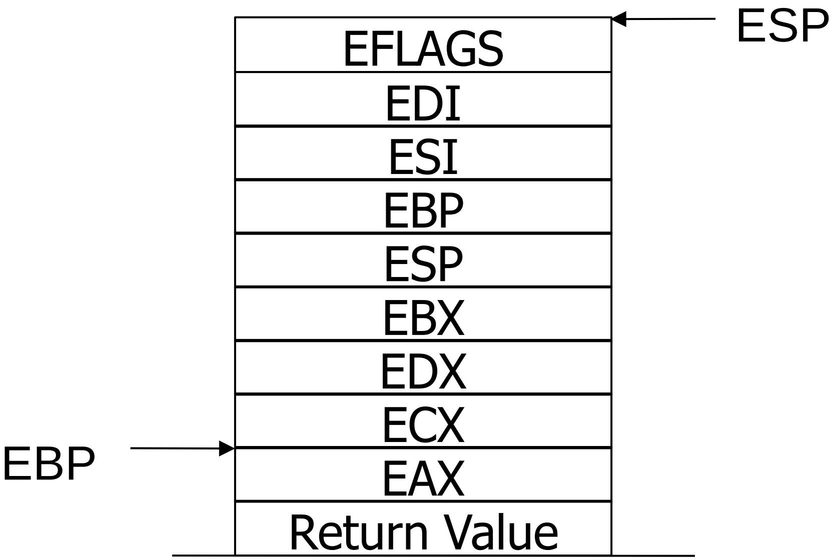
```
monitor:
            lea 16(%ebp), %edx
            movsbl   4(%edx), %eax
            xor      %edi, %edi
           testb    $4, %al
           jz       .NoIndex
            movsbq   6(%edx), %ecx
           negl     %ecx
           movl     (%ebp, %ecx, 4), %edi
           movsbl   7(%edx), %ecx
            imul     %ecx, %edi
```

Memory Trace Execution



CPU

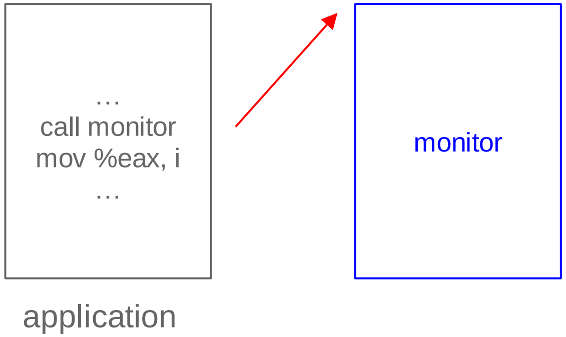
EAX: flags
EBX: ???????????????
ECX: - base reg
EDX: flags ptr
ESI: ?????????????
EDI: idx * scale + base
EBP: indirizzo eax orig.
ESP: ?????????????



EFLAGS
EDI
ESI
EBP
ESP
EBX
EDX
ECX
EAX
Return Value

```
monitor:
    .NoIndex:
        testb    $2, %al
        jz       .NoBase
        movsbq 5(%edx), %ecx
        negl     %ecx
        addl     (%ebp, %ecx, 4), %edi

    .NoBase:
        add       8(%edx), %edi
        movslq   (%edx), %esi

        push     %esi
        push     %edi
        call *16(%edx)
        addl     $8, %esp
```

Memory Trace Execution



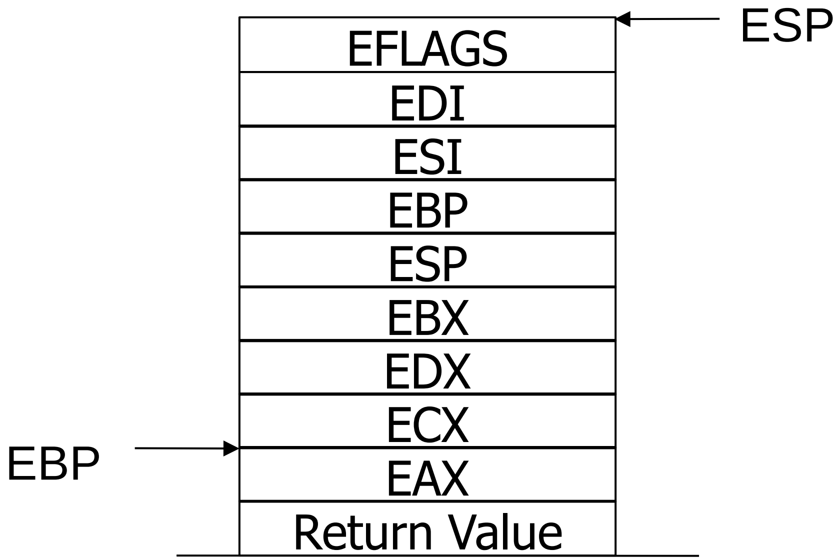
…
call monitor
mov %eax, i
…

monitor

application

CPU

EAX: flags	ESI: *size*
EBX: *?????????????????*	EDI: *bs+idx *scale+off*
ECX: *- base reg*	EBP: *indirizzo eax orig.*
EDX: *flags ptr*	ESP: *??????????????*



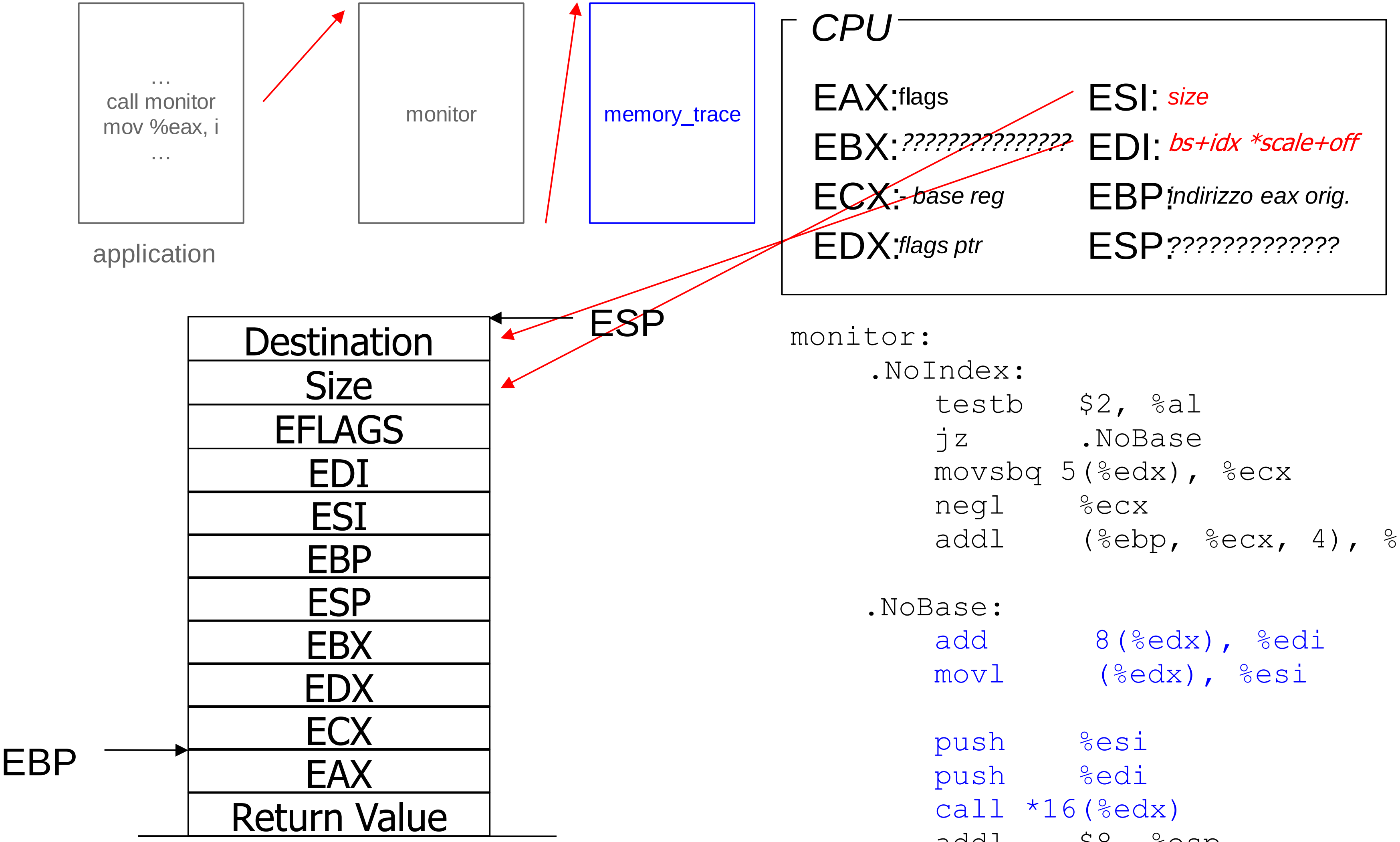
ESP

EFLAGS
EDI
ESI
EBP
ESP
EBX
EDX
ECX
EAX
Return Value

EBP

```
monitor:
     .NoIndex:
         testb    $2, %al
         jz       .NoBase
         movsbq 5(%edx), %ecx
         negl     %ecx
         addl     (%ebp, %ecx, 4), %edi

     .NoBase:
         add       8(%edx), %edi
         movl      (%edx), %esi

         push     %esi
         push     %edi
         call *16(%edx)
         addl     $8, %esp
```

Memory Trace Execution



…
call monitor
mov %eax, i
…

application

monitor

memory_trace

CPU

EAX: flags
EBX: *???????????????*
ECX: *base reg*
EDX: *flags ptr*

ESI: *size*
EDI: *bs+idx *scale+off*
EBP: *indirizzo eax orig.*
ESP: *?????????????*

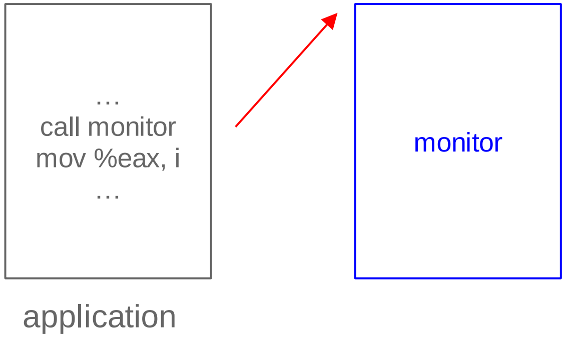
ESP

Destination
Size
EFLAGS
EDI
ESI
EBP
ESP
EBX
EDX
ECX
EAX
Return Value

EBP

```
monitor:
    .NoIndex:
        testb   $2, %al
        jz      .NoBase
        movsbq 5(%edx), %ecx
        negl    %ecx
        addl    (%ebp, %ecx, 4), %edi

    .NoBase:
        add      8(%edx), %edi
        movl     (%edx), %esi

        push    %esi
        push    %edi
        call *16(%edx)
        addl    $8, %esp
```

Memory Trace Execution



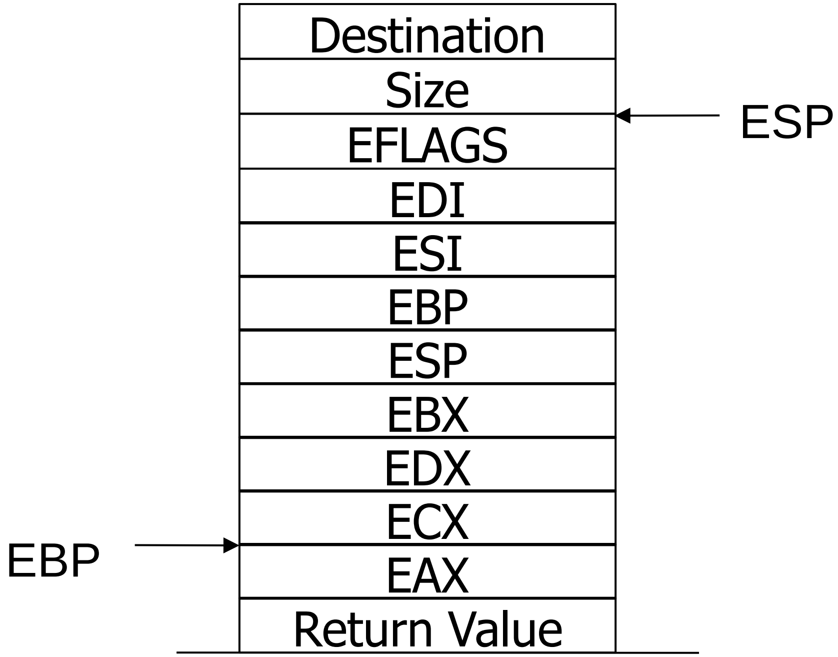
…
call monitor
mov %eax, i
…

application

monitor

CPU

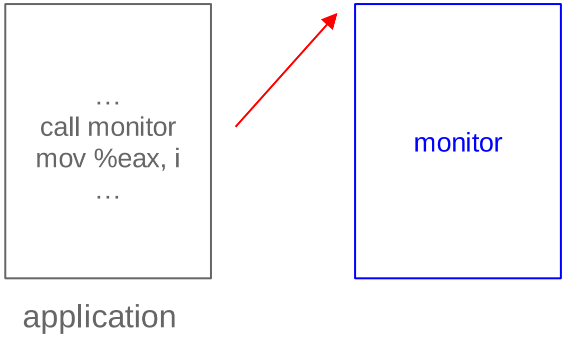
EAX:????????????? ESI: ?????????????
EBX:????????????? EDI: ?????????????
ECX:????????????? EBP:?????????????
EDX:????????????? ESP:?????????????



Destination
Size
EFLAGS
EDI
ESI
EBP
ESP
EBX
EDX
ECX
EAX
Return Value

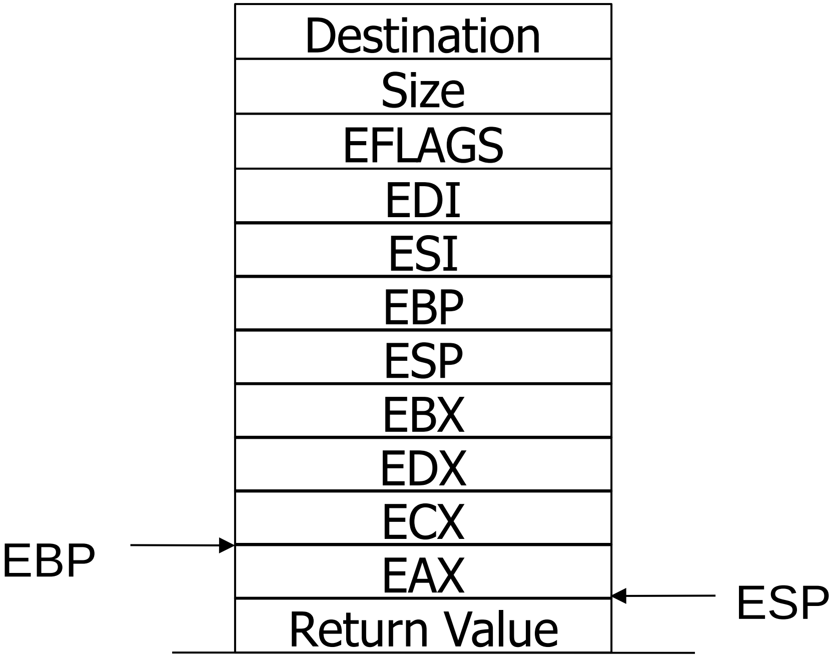
```
monitor:
    .NoIndex:
        testb    $2, %al
        jz       .NoBase
        movsbq 5(%edx), %ecx
        negl     %ecx
        addl     (%ebp, %ecx, 4), %edi

   .NoBase:
        add       8(%edx), %edi
        movl      (%edx), %esi

        push     %esi
        push     %edi
        call  *16(%edx)
        addl     $8, %esp
```

Memory Trace Execution



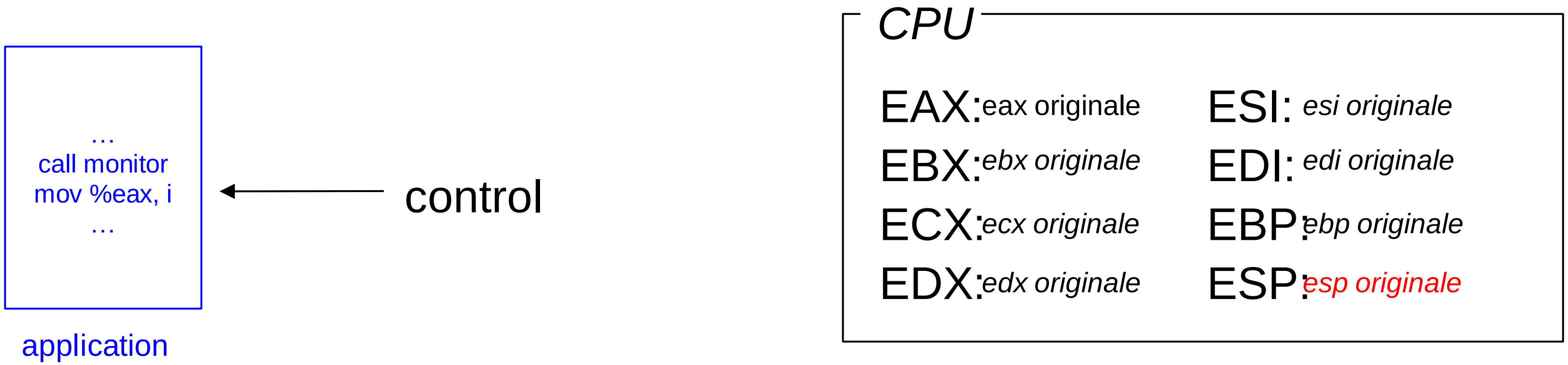
CPU

EAX: original eax
EBX: original *ebx*
ECX: original ecx
EDX: original *edx*
ESI: original *esi*
EDI: original *edi*
EBP: original *ebp*
ESP: *?????????????*



```
monitor:
         popfw
         pop     %edi
         pop     %esi
         pop     %ebp
         add     $4, %esp
         pop     %ebx
         pop     %edx
         pop     %ecx
         pop     %eax
         ret
```

Memory Trace Execution



…
call monitor
mov %eax, i
…

application

control

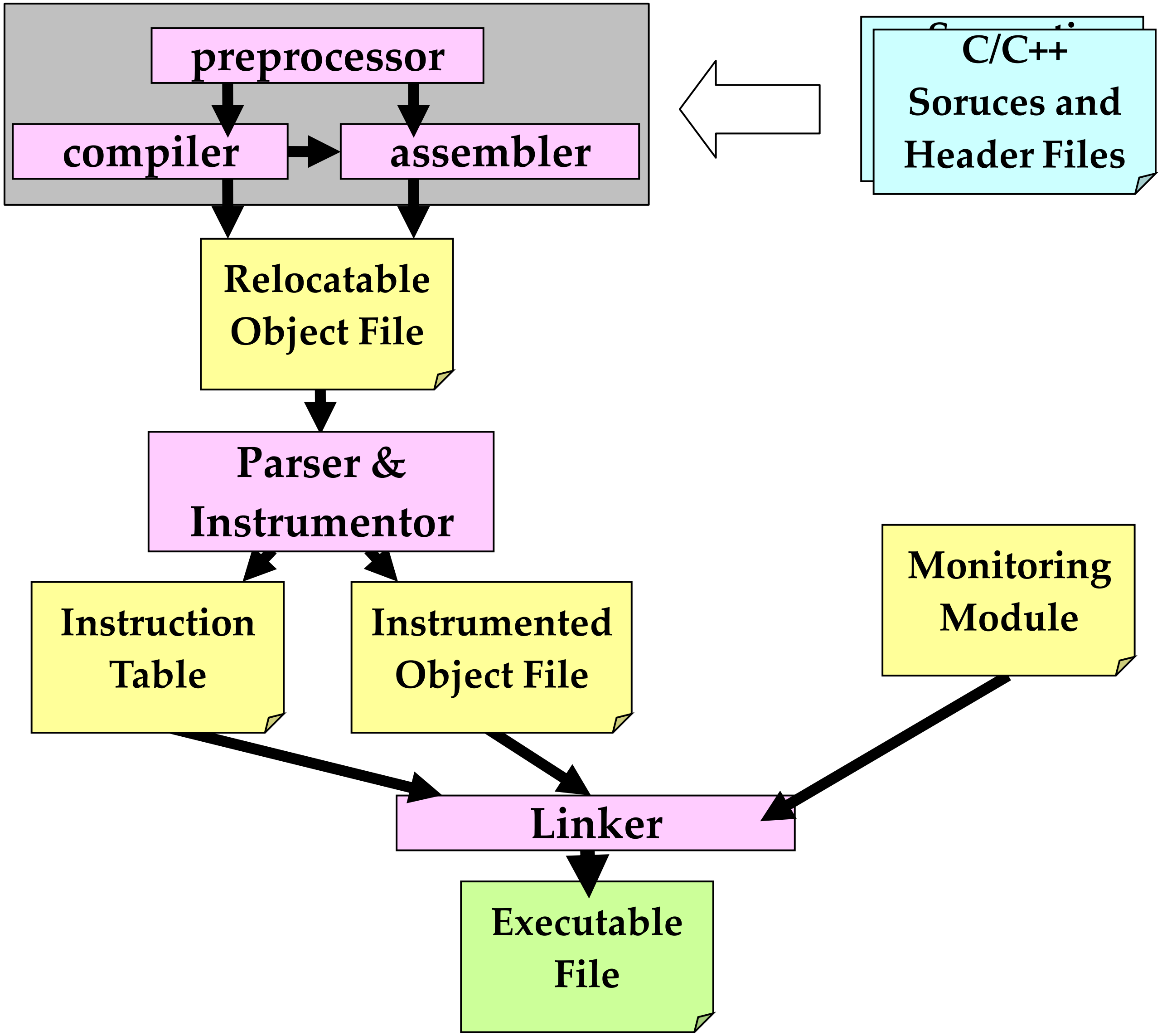
CPU

EAX: eax originale
EBX: *ebx originale*
ECX: *ecx originale*
EDX: *edx originale*

ESI: *esi originale*
EDI: *edi originale*
EBP: *ebp originale*
ESP: *esp originale*

```
monitor:
        popfw
        pop     %edi
        pop     %esi
        pop     %ebp
        add     $4, %esp
        pop     %ebx
        pop     %edx
        pop     %ecx
        pop     %eax
        ret
```

Summary



preprocessor

compiler

assembler

C/C++

Soruces and

Header Files

Relocatable

Object File

Parser &

Instrumentor

Instruction

Table

Instrumented

Object File

Monitoring

Module

Linker

Executable

File