



A. AMD64 SysV ABI

[...] the books of the Big-endians have been long forbidden, and the whole party rendered incapable by law of holding employments.

Jonathan Swift, *Gulliver's Travels*, I IV

We take a brief look at the Application Binary Interface (ABI), as defined in the System V specification for the AMD64/x86_64 architecture [59]. The ABI is used throughout the book, particularly in Parts II and III. Assuming you are already familiar with an ABI (e.g., ARM or RISC-V), we will only focus on the specifics of the AMD64 architecture. However, even if you are familiar with this architecture, you may be accustomed to a different assembly language syntax (e.g. AT&T vs. Intel), so you may still wish to review Section A.2.

A.1 Basic architecture

The basic architecture includes multiple CPUs that share access to memory and I/O address spaces. Each CPU contains several registers of different sizes. In this book, we will limit our discussion to the 16 integer registers: `rax`, `rcx`, `rdx`, `rbx`, `rsp`, `rbp`, `rsi`, `rdi`, `r8`, `r9`, ..., `r15`. We also add the instruction pointer (`rip`) and the flags register (`rflags`).

This architecture is the result of a long evolution that has managed to remain largely binary compatible with its earlier versions, beginning with the 16-bit Intel 8086 processor introduced in 1979 [62] and continuing with the 32-bit Intel 80386 processor announced in 1985 [46]. AMD introduced the 64-bit extension with the Opteron in 2003 [1], capitalizing on Intel's misstep when it tried to move away from this architecture and introduced the completely different Itanium in 2001 [95]. Intel finally adopted AMD's architecture with the Nocona in 2004 [51].

The CPU can interpret a wide range of machine instructions. These instructions are encoded in a highly complex manner using a variable number of bytes ranging from one to 16. For example, the instruction “push `rax`” (see Section A.1.3 below) is encoded with the single byte `50`. The lowest three bits encode the register; 0 is used for `rax`, 1 for `rcx`, and so on, with 7 representing `rdi`. For

instance, “push rbp” is encoded as 55.

- R** There are several ways in which we can observe the machine encoding of assembly instructions. However, since we need the pwntools [113] anyway, the easiest approach is to use the included `asm` tool, which is invoked as follows:

```
$ asm -c amd64 'push rax'
```

By default, this prints the resulting bytes in hexadecimal (even though octal would be better [43]). Try “pwn asm” instead of simply `asm` if the above command is not found.

As only three bits are reserved for encoding the register name, it is not possible to encode the registers `r8` through `r15`, introduced by AMD, directly. To encode these registers, the instruction must be prefixed with the *register extension (REX) prefix*, which provides space for the missing bit. For example, “push r9” is encoded as 41 51; the least significant bit of 41 and the three lower bits of 51 encode the 9 that selects `r9`.

Any byte in the 4x form is a REX prefix. In the 32-bit architecture, these bytes encode “inc register” and “dec register” instructions using a single byte. In 64-bit mode, these operations must be encoded using multiple bytes.

More bytes are required if the instruction involves constants or memory references (see Section A.2 below).

A.1.1 The address space

Although addresses in the memory address space are nominally 64 bits long, they are actually limited to 48 bits or 57 bits. This size is selected by a flag in a CPU system register, and for the purposes of this book we can assume that it is 48 bits. The upper 16 bits are unused and must be all equal to the most significant used bit (bit 47, counting from 0). Addresses in this form are called *normalized*. The address space is essentially split into two parts: a lower part whose addresses start with 0, and a higher part whose addresses start with 1. A large interval of unusable, unnormalized addresses lies between the two parts. The CPU raises an exception whenever an instruction tries to use an unnormalized address. The OS kernel intercepts the exception and terminates the offending process. For example, attempting to load an unnormalized address into `rip` will crash the process before `rip`’s contents are changed.

- R** When written in hexadecimal on 16 digits, normalized addresses must start with four zeros if the fifth digit is less than 8, and with four fs if the fifth digit is 8 or more. Any other configuration is unnormalized.

■ **Example A.1** Address 0xffffe00000000000 is normalized since its fifth digit is e \geq 8 and its first four digits are all f. Address 0x7ffd50072000 is also normalized. When written on 16 digits becomes 0x00007ffd50072000, its fifth digit is 7 < 8 and its first four digits are all zeros. ■

All memory addresses generated by the CPU, whether for fetching instructions or accessing operands, go through a Memory Management Unit (MMU) configured by the OS kernel when creating a process. The MMU’s main purpose is to translate the addresses before they access physical memory or memory mapped devices. This is not observable in Part II, but it becomes important in Part III.

One observable aspect is that every process has access only to the code, data, and stack assigned to it by the OS kernel. Some addresses (actually, most) are not *mapped* to anything. Accessing them causes a *page fault*, which, by default, crashes the process. Some addresses are mapped but marked as accessible only by the kernel. In Linux and most other operating systems, this is true for all the mapped addresses in the higher part of the address space (in our case, all normalized addresses that start with 4

fs when written on 16 hex digits). Some addresses are marked as read-only. This is true, for example, for the locations of memory that store instructions and constants.

R Initially we make the assumption that any readable location is also executable, i.e., the CPU can fetch instructions from any readable part of memory. This has been true for a long time in the history of the Intel architecture. We will revise this assumption in Chapter 9.

These mappings and protections have a *page-size* granularity, where the page size is usually 4 KiB. For example, if we know that address 0xab123 is mapped, then we know that *all* addresses in the range [0xab000, 0xabfff] are mapped too; moreover they must all be mapped with the same permissions.

A.1.2 Representing memory

Each byte in the memory address space has its own address, and most CPU instructions can read and update any accessible byte independently. However, instructions can also access memory in larger units, which Intel calls a *word* (two bytes), *double word* (or *dword*, four bytes) and *quad word* (or *qword*, 8 bytes). The latter is the default and most natural size for a 64-bit processor, and it is also the implicit size used when manipulating the stack.

The fact that the most natural size is not called *word* is, of course, an effect of the history of the architecture and the need to preserve compatibility with its earlier versions.

Since the qword is the preferred size for accessing memory, it make sense to represent memory as an array of rows, each containing eight bytes. Each row is aligned to eight, meaning it contains the 8 bytes whose addresses have the same quotient when divided by eight (or, in other words, the bytes whose addresses differ only in the lowest three bits).

R For the sake of simplicity, we also talk about 32-bit machines in Chapter 8. In those cases, we organize memory into rows of four bytes each. However, the rest of the conventions remain the same.

While this “array of rows” representation is mostly uncontroversial, we must now make two choices that may confuse those accustomed to a different convention.

- What is the order of the rows? Do the addresses increase by going up or down?
- What is the order of the bytes in each row? Do the addresses increase by going left or right?

Throughout the book, we will order the rows so that addresses increase *by going down*. Note that important documents, including the ABI specification itself, use the opposite convention. However, if we place lower addresses above, the stack top is actually at the top of the stack. Additionally, memory containing code can be read most naturally from top to bottom. Data structures, such as C `structs`, have their fields ordered in the same way that they are declared. Most importantly, this is how memory appears in the debugger, which settles the issue for us.

However, the order of bytes in each row is more controversial, and our choice is sometimes unnatural. Specifically, we will order the bytes of each row such that the addresses increase *from right to left*. Again, this is mostly forced upon us by the way the debugger shows memory, particularly the stack and the heap, when organizing it in qwords. Since AMD64 is *little-endian*, the lowest address byte of each qword is also the least significant byte in the qword, and therefore ends up on the right.

You may have heard it multiple times by now, but in case you missed it, the names *little-endian* and *big-endian* come from a paper by Danny Cohen [26]. In the paper, Cohen satirized the controversies about the best ways to send words on a serial line, by paralleling them to the silly war between Lilliput and Blefuscu about how to eat hard-boiled eggs (i.e., starting from

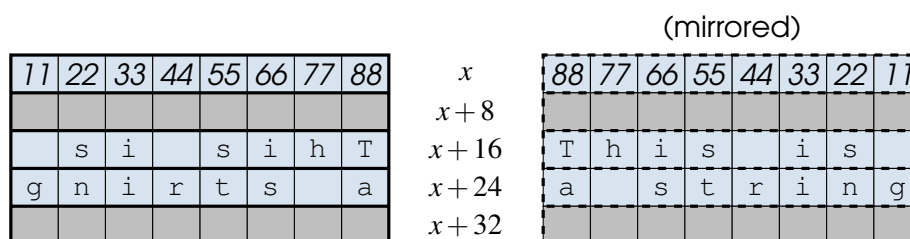


Figure A.1 – Memory representation

the little end or the big end).

Note that our ordering convention has some annoying consequences. Within each row, machine instructions and character strings are shown backwards. This is especially awkward for strings, so we try to mitigate the confusion by representing the same part of memory in two ways: right-to-left and left-to-right. The latter representation uses dashed lines and is labelled “mirrored”. Figure A.1 shows 5 rows of memory, spanning addresses $[x, x+40)$, according to these conventions. The representation on the left is the one we prefer. The “mirrored” representation on the right is used occasionally, particularly when we are interested in the strings stored in memory. In the diagram on the left, the first byte of each row is the rightmost one, whereas in the diagram on the right, the first byte of each row is the leftmost one. The address of the first byte of each row is shown in the middle. A qword with value `0x1122334455667788` is stored at address x , as is the string “This is a string” at address $x+16$. Note that the qword can be read correctly in the representation on the left, and is shown in reverse in the mirrored representation on the right. The opposite is true for the string.

In a big-endian architecture we could have used the representation on the right exclusively. However, big-endianness has gone out of fashion and is not used by any modern CPU architecture. Things would be different if we were talking of network packet headers, since the layout of most of them, including those for UDP, TCP, and IP, is big endian.

A.1.3 The stack

The sixteen integer registers are mostly equivalent, but several instructions use a few of them for specific purposes. The most relevant one for us is `rsp`, the “stack pointer”, which is implicitly used by instructions such as `push`, `pop`, `call` and `ret` (see Section A.2.3 below). These instructions assume that `rsp` points to the top of the stack and update it to add (`push` and `call`) or extract (`pop` and `ret`) items.

Adding an item to the top of the stack involves subtracting 8 from `rsp` and then writing to the resulting address. In our representation, this is visualized as follows:



Extracting an item from the top of the stack involves reading from the address stored in `rsp` and then adding 8 to `rsp`. This can be visualized as follows:



The `rsp` register can point anywhere, and the instructions that use it don't care as long as the corresponding memory is readable and, in the case of `push` and `call`, also writable. Conversely, keep in mind that the stack is just memory and there is no obligation to access it exclusively through `rsp`.

Besides its implicit use by the aforementioned instructions, `rsp` can be used as a general-purpose register. It can be the source or a destination register in `mov`, `add`, `sub`, etc., and it can also be used as a base register in instructions that access memory (Section A.2.1.1).

A.2 Intel Assembly syntax

We make a quick tour of the things that we might encounter in the course, without any pretense of completeness.

A.2.1 Operational instructions

Operational instructions can use one or two explicit operands. Some instructions use implicit operands exclusively, while others use them in addition to explicit operands.

A typical instruction with two explicit operands is written in assembly as follows:

```
opcode op1, op2
```

Where *op₁* is either the name of a register, or a *memory reference* (see Section A.2.1.1 below); *op₂* can be a register, a memory reference, or a constant. In all cases, the instruction takes its operands from *op₁* and *op₂* (the contents of the register, the contents of memory, or the constant), and computes a result. This result is then written in *op₁* (in either the register, or memory). In most cases, the instruction also computes some additional information about the result and updates the `rflags` register as an implicit operand. In all cases, the `rip` register is incremented by the size of the instruction, thus allowing it to point to the next instruction. For most purposes, we can assume that this increment is performed when the instruction starts, meaning that `rip` is already pointing to the next instruction while the current instruction is being executed. This is important to know when offsets are added to `rip`, as in A.2.1.1 below, or when `rip` is saved, e.g. by `call`.

■ **Example A.2** Consider the following instruction

```
add rax, rdx
```

This computes the sum of the contents of the 64-bit binary values stored in `rax` and `rdx` and replaces the contents of `rax` with the result. Furthermore, it computes a set of *flags* from the result and updates the corresponding bits in the `rflags` registers. Some interesting flags are:

- ZF (Zero Flag): set iff the result is zero;
- CF (Carry Flag): set iff the addition has caused a carry (unsigned overflow);
- OF (Overflow Flag): set iff the addition has caused a signed overflow.

The 64-bit numbers being added can be either unsigned or two's complement signed, since the hardware is the same for both representations. The only difference is in the way overflow is detected. Accordingly, the CPU computes the CF and OF flags for both cases simultaneously, and the software must know which to check.

The following instruction uses a constant:

```
add rax, 0x1122
```

This adds 0x1122 to the contents of `rax` (and updates `rflags`). This instruction is encoded as `48 05 22 11 00 00`. We can identify the `48` REX prefix and the constant, which is encoded in little-endian format over four bytes at the end of the instruction. The REX prefix is necessary here because, unlike `push` and `pop`, this instruction defaults to 32-bit operands and the REX prefix switches it to 64-bit operands. Without the prefix, the resulting `05 22 11 00 00` bytes would encode “`add eax, 0x1122`”.

■

The most common operational instruction is `mov`, that copies the second operand into the first one. It doesn't update the `rflags` register.

A.2.1.1 Memory references

Memory references can be used for either the first or the second explicit operand, but not both. They are written as follows:

```
size PTR [expression]
```

where *size* is one of `BYTE`, `WORD`, `DWORD` or `QWORD`. The *expression* within the square brackets should compute a memory address and can take several forms. The most complex form is as follows:

$$offset + base + index * scale$$

where *offset* is an 8-bit or 32-bit signed constant; *base* and *index* are two registers; and *scale* is one of 1, 2, 4 or 8. The addends can be written in any order, and “`* 1`” can be omitted. The other forms can be obtained from this one by omitting one or two addends, in any combination.

■ **Example A.3** The instruction

```
mov rax, QWORD PTR [0x10 + rbx + rcx * 8]
```

computes the expression in the square brackets to obtain an address *x*, then, reads 8 bytes from memory starting at address *x*, and finally, stores the bytes in `rax` in little-endian order: the byte read from *x* goes in the least significant part of `rax` and the byte read from *x* + 7 goes into the most significant part.

The instruction

```
add WORD PTR [0x10 + rbx + rcx * 8], 1
```

will compute the address *x* as before. It will then read a 16-bit binary number from addresses *x* (LSB) and *x* + 1 (MSB), add one to it and store the result in memory at addresses *x* (LSB) and *x* + 1 (MSB). Finally, it will update `rflags`.

The address expression can use fewer than three addends and the addends can be written in any order. For example, consider the instruction:

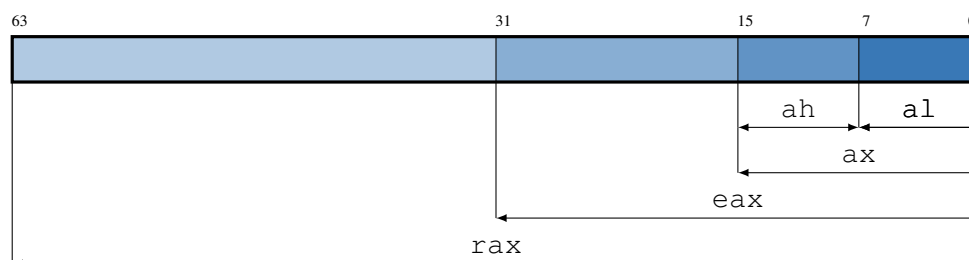


Figure A.2 – Partial registers

```
mov QWORD PTR [rbp - 0x18], rdi
```

The expression “`rbp - 0x18`” should be interpreted as “ $-0x18 + rbp + 0$ ”, i.e., it consists of a (negative) offset and a base. The “*index * scale*” addend is omitted and replaced by 0. The instruction copies the qword stored in `rdi` in memory, at the address obtained by subtracting `0x18` from the contents of `rbp`. The LSB of `rdi` will go at address x and the MSB will go at address $x + 7$. ■

There is also a “rip-relative” addressing form, written as follows:

```
size PTR [rip + offset]
```

In this case the address is obtained by adding *offset* to the address of the next instruction (i.e., the value stored in `rip` when the above instruction is executed). This is merely an encoding trick designed to partially overcome the 32-bit limitation of the *offset* field in the other addressing forms, but it has the useful side effect of making the instruction position-independent (see B.3.1).

■ **Example A.4** Consider the following instructions:

```
0x401000:    mov rax, QWORD PTR [rip + 0x101]
0x401007:    ret
```

The first column shows the addresses of the instructions. The first instruction computes the address $x = 0x101 + 0x401007 = 0x401108$. Then, it reads a qword from address x and stores it in `rax`.

Assume that x is the address of a variable V which is part of the program. If we load the *entire* program at a different address, the `mov` instruction will still work. For example, assume that we shift all addresses by `0x500000`. In this case, the two instructions above would end up at addresses `0x901000` and `0x901007` respectively. When the CPU computes “`rip + 0x101`” from the new position, it obtains $0x101 + 0x901007 = 0x901108 = x + 0x500000$ —the correct new address of V . ■

A.2.1.2 Partial registers

There is a somewhat limited and irregular support for accessing *subsets* of the integer registers. AMD has tried to make it more uniform, but irregularities persist. In general, instructions can access the least significant byte, word and dword of each register. The names of these subsets are as follows:

- byte: `al`, `cl`, `dl`, `bl`, `spl`, `bpl`, `sil`, `dil`, `r8b`, `r9b`, ..., `r15b`;
- word: `ax`, `cx`, `dx`, `bx`, `sp`, `bp`, `si`, `di`, `r8w`, `r9w`, ..., `r15w`;
- dword: `eax`, `ecx`, `edx`, `ebx`, `esp`, `ebp`, `esi`, `edi`, `r8d`, `r9d`, ..., `r15d`.

The second byte of `rax`, `rcx`, `rdx` and `rbx` can be accessed using the `ah`, `ch`, `dh` and `bh` names, but this is not possible for the other registers. Figure A.2 shows the partial registers of the `rax` register.

Writing into a byte- or word-sized subset of a register leaves the rest of the register unaltered, but writing into a dword-sized subset zeroes out the other 32 bits.

A.2.2 Stack instructions

We can push an item on top of the stack with

```
push source
```

where *source* can be a constant, a register, or a memory reference. Note that in the special case of “push `rsp`”, the value that is pushed on the stack is the value that `rsp` had *before* the decrement. Moreover, if the source is a memory reference that uses `rsp` in the address expression, the value used is also the one before the decrement.

We can pop an item from the top of the stack with

```
pop destination
```

where *destination* can be a register or a memory reference. In the special case of “pop `rsp`”, the final value of `rsp` is the value extracted from the top of the stack. If the destination is a memory reference that uses `rsp` in the address expression, the value used is the one before the increment.

A.2.3 Control instructions

The most important instructions that can redirect the control flow are conditional and unconditional jumps, `call` and `ret`. Unconditional jumps are written as follows:

```
jmp offset
```

Where *offset* is a signed 8-bit or 32-bit constant. The instruction adds *offset* to `rip`. Note that this is another form of `rip`-relative addressing, even if `rip` is not mentioned explicitly.

Jumps can also be *indirect*, either through a register or through memory. In this case they are written as follows:

```
jmp register  
jmp QWORD PTR [expression]
```

Both instructions copy their operand (taken from the register or from memory) into `rip`. Note that the address is copied, not added, so these forms of jumps need absolute addresses.

Conditional jumps are written as follows:

```
jCC offset
```

Where *offset* is used as above, and *CC* is a *Condition Code* that must be tested on `rflags`. For example, `jz` jumps (adds *offset* to `rip`) only if the `ZF` is set, while `jnz` does the opposite. The normal way to set the flags is by using a `cmp` instruction, whose purpose is to compare two values. However, the flags are also set by all the arithmetic and logic instructions, and also by other specialized instructions.

Subroutine calls are supported by the `call` and `ret` instructions, that use the stack implicitly (more precisely, they use `rsp` implicitly):


```
call offset
```

This instruction pushes `rip` on the stack, then adds *offset* to `rip`.

- R** Since we are going to abuse this stuff a lot, it is helpful to review what the instruction does more precisely: it subtracts 8 from `rsp`, stores the current value of `rip` (which is pointing at the instruction that follows the `call` in memory) at the resulting address, then adds the *offset* to `rip`. Execution continues from the new `rip`.

Indirect `calls` are also possible, using the same syntax as for indirect `jumps`:

```
call register  
call QWORD PTR [expression]
```

Take note of the relative/absolute nature of the addresses involved: “`call offset`” uses a `rip`-relative address for the destination of the jump, but it saves the current *absolute* value of `rip`. Indirect `calls` need absolute addresses also for the destination of the jump.

Finally, the `ret` instruction can be used with no explicit operands:

```
ret
```

This instruction pops an item from the stack and copies it into `rip`.

- R** Review it carefully. The instruction copies the `qword` stored at `rsp` into `rip`, then adds 8 to `rsp`. Execution continues from the new `rip`.

Again, since the full contents of `rip` are replaced, the address stored on top of the stack must be absolute. If everything works as intended, a `ret` instruction placed at the end of a subroutine will recover the address saved by the `call` that entered the subroutine, and thus will return to the caller.

A.2.4 Labels

In the above instructions, we have used the numerical offsets because this is what is actually encoded in the instructions. When disassembling a binary, or debugging a program, we may have to live with these offsets, if the disassembler or debugger has no other information. If *symbols* are available (see Section A.4.1.2), the disassembler and the debugger will use them as effectively as possible. If an address exactly equals the value of a symbol, the tools will display the symbol either instead of the address, or as a comment. If there is no exact match, the tools will find the nearest symbol and print expressions such as “`main + 0x17`”, which can be interpreted as “0x17 bytes after the start of `main`”. While these expressions are often useful, they can sometimes be confusing. Remember that they do not necessarily imply that the address has anything to do with the symbol; it may just be that the symbol was the first one found when moving backwards from the address.

In the case of `rip`-relative addressing, the tools typically show the absolute address resulting from adding the offset to the address of the next instruction, which is usually much more useful than the plain offset.

When *writing* assembly code, we can use *labels* to give addresses symbolic names and then refer to them using these names. The assembler knows when to use the address as it is and when to compute offsets from `rip` based on the instruction at which the label is used.

■ **Example A.5** Run the following:

```
$ asm -c amd64 'jmp forward; add rax, 0x1122; forward: push rax'
```

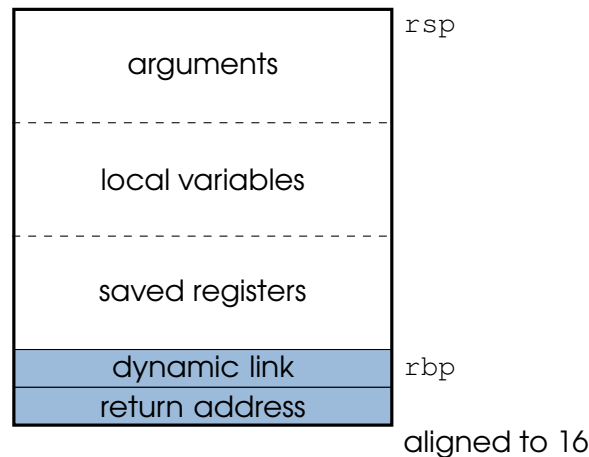


Figure A.3 – The standard stack frame

This assembles a sequence of three instructions: a jump to a label called `forward`, the second `add` instruction from Example A.2, and a `push` instruction. We have defined the `forward` label just before the `push` instruction. Syntactically, the definition consists of an identifier followed by a colon. The label represents the address of whatever follows its definition. In this case, it is the address of the “`push rax`” instruction. The output of the `asm` command is `eb 06 48 05 22 11 00 00 50`. We can visualize it as follows, assuming `0x401000` as an arbitrary address for the first instruction:

<code>0x401000:</code>	<code>eb 06</code>	jmp	<code>0x401008</code>
<code>0x401002:</code>	<code>48 05 22 11 00 00</code>	add	<code>rax, 0x1122</code>
<code>0x401008:</code>	<code>50</code>	push	<code>rax</code>

Following the first two bytes (`eb 06`), we see the encoding of the `add` instruction from Example A.2 (`48 05 22 11 00 00`) immediately followed by the encoding of “`push rax`” (`50`). In this example, the `forward` label has the value `0x401008`. The first two bytes encode the “`jmp forward`” instruction: `eb` is the opcode and `06` is the number of bytes that the jump instruction must add to `rip` to skip the `add` instruction. The assembler knows that `jmp` uses `rip`-relative addressing, so it has automatically calculated `forward - 0x401002 = 6`. Note the result doesn’t depend on the assumption we made on the address of the first instruction. ■

A.3 Function calling sequence

We limit our discussion to functions that accept only integer and pointer arguments.

Each function has its own *stack frame* which has room for local variables, actual arguments, and control information. The most important piece of control information is the return address. A new stack frame is allocated on the stack when a function is entered, and is deallocated when the function terminates. Figure A.3 shows the general shape of a stack frame. From bottom to top, we encounter the following:

- the return address;
- the dynamic link (see below);
- (optional) room for saving registers (see the discussion about scratch registers below);
- (optional) room for the variables declared inside the function (local variables);
- (optional) room for the first six actual arguments.

During function execution, the `rbp` register (also known as the *base register* or *frame pointer*) is reserved as a pointer to the base of the current stack frame. The function accesses its local variables and its first six arguments using negative constant offsets from the frame pointer, with memory references such as “[`rbp` - *offset*]”.

The ABI mandates that the address just below the start of the frame or, equivalently, the address stored in `rbp`, must be a multiple of 16.

The caller must pass the first six arguments using the registers `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9`, in this order. Each argument consumes a register, even if the argument is smaller than 64 bits. If the function accepts more than six arguments, the caller must push the additional arguments onto the stack in *reverse* order (the seventh argument is on top, the eighth argument is below it, and so on). As with registers, each argument consumes an 8-bytes stack line, even if the argument is smaller than 64 bits. These arguments go below the return address and are not considered part of the new stack frame.

If the function returns a value, it should leave it in the `rax` register.

This seemingly erratic order of registers is the result of numerous experiments and technical requirements [44]. For example, the first two registers, `rdi` and `rsi`, are implicitly used by the string instructions, such as `movs`, since 8086 (`di` stands for Destination Index and `si` for Source Index). These instructions are used to optimize common functions such as “`memcpy(dst, src, n)`”. If we pass the `dst` parameter in `rdi` and the `src` parameter in `rsi`, they are already in the right places for `movs`. However, note that `movs` requires the number of bytes to be copied in `rcx`, suggesting the use of `rcx` as the third register in the sequence. The ABI convention uses `rdx` instead due to competing constraints.

A.3.1 Prologue and epilogue

The creation of the stack frame begins with the `call` instruction in the caller. This pushes the return address onto the stack and yields control to the function. The function then completes the frame with a standard *prologue* consisting of the following instructions:

```
push rbp
mov  rbp, rsp
```

The first instruction saves the base pointer of the caller function and the second one loads the new base pointer. The previous base pointer is also called the *dynamic link*, because it is a pointer to the stack frame that precedes the current one during execution.

The function can freely use a subset of the CPU registers, called *scratch registers*. These are the six registers used for parameter passing, plus `rax`, `r10` and `r11`. The function can use the other registers, too, but it must restore their original contents before returning to the caller. This is accomplished by pushing the used registers after the prologue and popping them before the epilogue.

After the prologue and the pushes of the used non-scratch registers, if applicable, an instruction like this can be found:

```
sub  rsp, n
```

This instruction allocates *n* bytes on the stack, making room for the local variables and the arguments. The ABI does not specify how the function should allocate its variables within this space.

At the end of the function, we find the standard *epilogue*, which consists of the following instructions:

```

mov rsp, rbp
pop rbp
ret

```

These instructions restore the state of `rsp` and `rbp` and then yield control back to the caller. Sometimes the first two instructions are replaced by the single `leave` instruction, which accomplishes the same task.

■ **Example A.6** Let's consider the following C code, contained in file `foo.c`:

```

1 long foo(long a, long b)
2 {
3     long c = a + b;
4     return c;
5 }
6
7 int main()
8 {
9     long x = foo(3, 4);
10    return x;
11 }

```

Compile it using the following command:

```

$ gcc -o foo -mno-red-zone -zexecstack -no-pie -znorelro \
    -fcf-protection=none foo.c

```

R The `-mno-red-zone` option disables the *red zone* (see Section A.3.2.2 below); the other three options disable some features that are examined elsewhere in the book: non-executable stack (Section 9.1), PIE (Section 9.6), RELRO (Section 10.3) and Control Flow Integrity (Section 10.7).

We obtain the `foo.o` file, that we can disassemble as follows:

```

$ objdump -d -Intel foo.o

```

R The `-d` option is for disassembly. The `-Intel` option selects the Intel syntax instead of the default AT&T syntax.

The output contains a lot of code coming from other object files that are automatically linked by `gcc`. The part of the output pertaining to the `main` function should look like this:

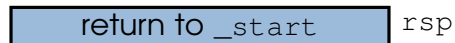
```

00000000040112b <main>:
40112b:      55                push    rbp
40112c:      48 89 e5          mov     rbp, rsp
40112f:      48 83 ec 10       sub     rsp, 0x10
401133:      be 04 00 00 00    mov     esi, 0x4
401138:      bf 03 00 00 00    mov     edi, 0x3
40113d:      e8 c4 ff ff ff    call    401106 <foo>
401142:      48 89 45 f8       mov     QWORD PTR [rbp-0x8], rax
401146:      48 8b 45 f8       mov     rax, QWORD PTR [rbp-0x8]
40114a:      c9                leave

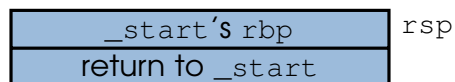
```

```
40114b:      c3                      ret
```

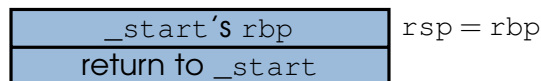
Let's review the instructions. We recognize the standard prologue at addresses 0x40112b and 0x40112c. This is followed by a `sub` instruction that allocates 0x10 (16) bytes. By the time execution reaches `main()`, a lot of code has already been executed, and the stack already contains a lot of data. For now, we can ignore all of this and assume that `main()` has been called by `_start`, the entry point of the program. Now, let's examine the effect of these instructions on the stack. Immediately before executing “`push rbp`”, the top of the stack contains the return address inside `_start`, and `rbp` (not shown) contains the frame pointer used by `_start`:



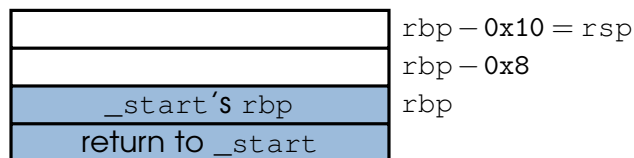
The “`push rbp`” instruction decrements `rsp` by 8, then stores the current value of `rbp`:



The “`mov rbp, rsp`” instruction initializes `main's` frame pointer, by copying the current value of `rsp`:



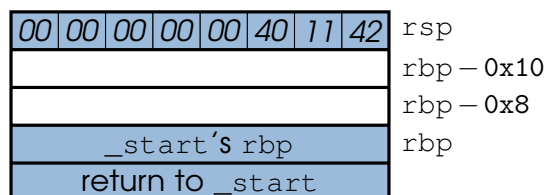
The “`sub rsp, 0x10`” instruction subtracts 0x10 from `rsp`, thus allocating space for `main's` local variables:



This is the complete stack frame of `main`, even though we don't know how the top two stack lines will be used. Now `main` starts the function calling sequence for `foo`. As expected, the next two instructions load the actual arguments 3 and 4 into the `rdi` and `rsi` registers.

The compiler has used instructions that write to the lower 32-bits of `rdi` and `rsi`, taking advantage of the fact that the CPU will zero out the upper bits. In this way it can spare the REX prefix for these two instructions.

The `call` instruction pushes the address of the next instruction, 0x401142, onto the stack. This begins the construction of `foo's` stack frame on top of `main's` stack frame.



Now execution continues in `foo`. Let's take a look at its instructions, in the output of `objdump`:

```

0000000000401106 <foo>:
401106:      55                push    rbp
401107:     48 89 e5          mov     rbp, rsp
40110a:     48 83 ec 20       sub     rsp, 0x20
40110e:     48 89 7d e8       mov     QWORD PTR [rbp-0x18], rdi
401112:     48 89 75 e0       mov     QWORD PTR [rbp-0x20], rsi
401116:     48 8b 55 e8       mov     rdx, QWORD PTR [rbp-0x18]
40111a:     48 8b 45 e0       mov     rax, QWORD PTR [rbp-0x20]
40111e:     48 01 d0          add     rax, rdx
401121:     48 89 45 f8       mov     QWORD PTR [rbp-0x8], rax
401125:     48 8b 45 f8       mov     rax, QWORD PTR [rbp-0x8]
401129:     c9                leave
40112a:     c3                ret

```

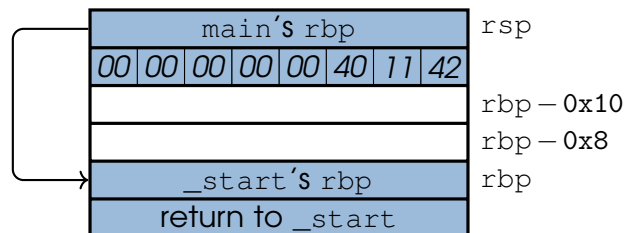
The `foo` function begins at address `0x401106`. Let's take another look at the `call` instruction in `main`:

```
40113d:      e8 c4 ff ff ff    call    401106 <foo>
```

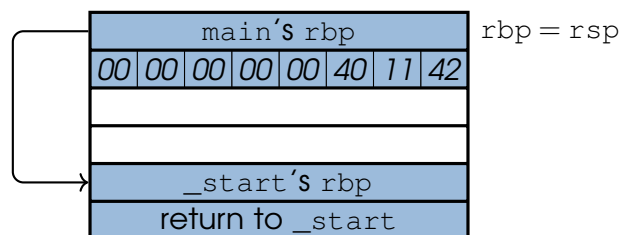
From the disassembly, it appears that the `call` contains the address of `foo`. However, examining the bytes that encode the instruction reveals the offset `0xfffffc4`, equivalent to `-0x3c` in 2's complement. This is `0x40113d + 5 - 0x401106`, i.e., the constant that must be added to the address of the instruction following the `call` to obtain the address of `foo`. As we anticipated in Section A.2.4, `objdump` computed the resulting address for us. Furthermore, since `objdump` knows that symbol `foo` is at that address, it has also correctly printed the name of the function that is being called.

We will now review the instructions of `foo`. We identify the standard prologue at addresses `0x401106` and `0x401107`. Immediately after the prologue at address `0x40110a`, we find the instruction that allocates `0x20` (32) bytes on the stack. These bytes provide storage space for `a`, `b` and `c`. Let's examine this in more detail.

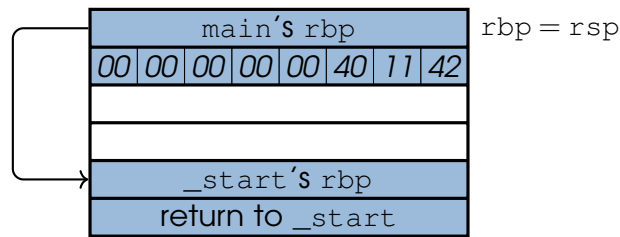
The “`push rbp`” instruction saves the frame pointer of `main`, creating the dynamic link between the stack frames of `foo` and `main`.



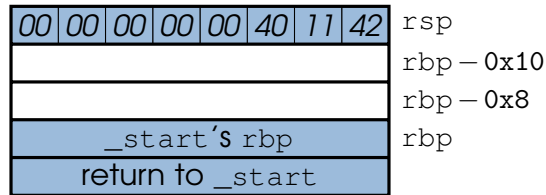
The “`mov rbp, rsp`” instruction initializes the stack pointer of `foo`:



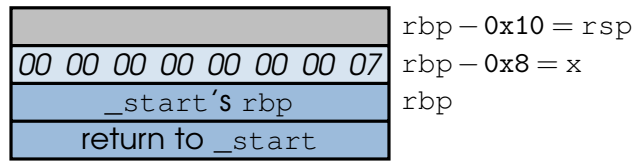
The “`sub rsp, 0x20`” instruction allocates 32 bytes on the stack:



The “pop rbp” instruction restores the frame pointer of main:



Now the stack is in the state that it was when we started the execution of `foo`, immediately after the `call`. The `ret` instruction pops `0x401142` and jumps there. After the `ret`, the stack frame of `main` is again on top of the stack. The next instruction reveals that `x` is stored at `rbp-0x8`:



The top line is unused. The reason is the same as for `foo`. The instruction at address `0x401129` copies `x` into `rax`, to pass the return value to `_start` (which will pass it to the kernel). We have finally reached the epilogue of `main`, which will dismantle the stack frame and yield control to `_start`. ■

A.3.2 Optimizations

The compiler may simplify the standard prologue and epilogue, and the other instructions described above, in several ways. Some optimizations are enabled by default, while some others are activated only when we explicitly ask for optimized code (option `-O` followed by 1, 2 or 3). Some optimization have their own options and can be enabled individually.

A.3.2.1 Omitting the frame pointer

This optimization is enabled by `-fomit-frame-pointer` and is also selected by `-O1` and higher. When this option is enabled, the compiler uses `rsp` as the frame pointer, thus removing the need for saving, loading and restoring `rbp`. With this optimization enabled, the standard prologue disappears and the standard epilogue becomes a simple `ret`.

However, `rsp` may need to be adjusted to align it to 16. Moreover, before returning, it needs to be restored with an `add`. This can be sometimes avoided if we also use the red zone.

Another useful effect of this optimization is that the `rbp` register is free to be used for register allocation (see below).

■ **Example A.7** This is the disassembly of function `foo` from Example A.6 compiled with the same options, plus `-fomit-frame-pointer`:

```

0000000000401106 <foo>:
401106:      48 83 ec 20      sub     rsp,0x20
40110a:      48 89 7c 24 08    mov     QWORD PTR [rsp+0x8],rdi
40110f:      48 89 34 24      mov     QWORD PTR [rsp],rsi
401113:      48 8b 54 24 08    mov     rdx,QWORD PTR [rsp+0x8]
401118:      48 8b 04 24      mov     rax,QWORD PTR [rsp]
40111c:      48 01 d0          add     rax,rdx
40111f:      48 89 44 24 18    mov     QWORD PTR [rsp+0x18],rax
401124:      48 8b 44 24 18    mov     rax,QWORD PTR [rsp+0x18]
401129:      48 83 c4 20      add     rsp,0x20
40112d:      c3                ret

```

The standard prologue has disappeared and `rsp` is restored with an `add`, since `rbp` no longer contains its original value. ■

A.3.2.2 The red zone

The ABI guarantees that each function has 128 bytes available above `rsp`, called the red zone. If the function doesn't call other functions, it can use this space for its arguments and local variables, without any need to adjust `rsp`. This optimization is enabled by default, and can be disabled with the option `-mno-red-zone`.


■ **Example A.8** This is the disassembly of function `foo` from Example A.6, compiled using the same options with the exception of `-mno-red-zone`:

```

0000000000401106 <foo>:
401106:      55                push    rbp
401107:      48 89 e5          mov     rbp, rsp
40110a:      48 89 7d e8        mov     QWORD PTR [rbp-0x18],rdi
40110e:      48 89 75 e0        mov     QWORD PTR [rbp-0x20],rsi
401112:      48 8b 55 e8        mov     rdx,QWORD PTR [rbp-0x18]
401116:      48 8b 45 e0        mov     rax,QWORD PTR [rbp-0x20]
40111a:      48 01 d0          add     rax,rdx
40111d:      48 89 45 f8        mov     QWORD PTR [rbp-0x8],rax
401121:      48 8b 45 f8        mov     rax,QWORD PTR [rbp-0x8]
401125:      5d                pop     rbp
401126:      c3                ret

```

Note that `rsp` and `rbp` are equal and the instructions at addresses `0x401107`, `0x40110a`, and so on, access memory above `rbp`, and therefore above `rsp`. This is dangerous in general, since a signal handler that interrupts this function may use the same memory for its own purposes. However, the ABI guarantees that signal handlers will use the stack only starting from address `rsp-128`.

 By “above” we mean “above in the page”, in our memory representation. The addresses are numerically *below* `rsp`.

If we combine the red zone with `-fomit-frame-pointer`, we obtain:

```

0000000000401106 <foo>:
401106:      48 89 7c 24 e8    mov     QWORD PTR [rsp-0x18],rdi
40110b:      48 89 74 24 e0    mov     QWORD PTR [rsp-0x20],rsi
401110:      48 8b 54 24 e8    mov     rdx,QWORD PTR [rsp-0x18]

```

```

401115:      48 8b 44 24 e0      mov     rax,QWORD PTR [rsp-0x20]
40111a:      48 01 d0            add     rax,rdx
40111d:      48 89 44 24 f8      mov     QWORD PTR [rsp-0x8],rax
401122:      48 8b 44 24 f8      mov     rax,QWORD PTR [rsp-0x8]
401127:      c3                ret

```

Now there is no prologue, and the epilogue is just a `ret` instruction. ■

A.3.2.3 Register allocation

Registers are the fastest storage available to the CPU, so one of the most effective optimization implemented by compilers is to store values in registers in such a way to minimize accesses to memory. If register allocation is enabled (`-O1` and higher), the compiler will most likely not copy on the stack the arguments passed via registers, and will use the registers directly.

■ **Example A.9** If we compile `foo.c` of Example A.6 with `-O1`, we obtain the following:

```

0000000000401106 <foo>:
401106:      48 8d 04 37          lea     rax,[rdi+rsi*1]
40110a:      c3                ret

000000000040110b <main>:
40110b:      b8 07 00 00 00      mov     eax,0x7
401110:      c3                ret

```

The `foo` function has been reduced to a single instruction, besides the necessary `ret`. The compiler has used the contents of `rdi` and `rsi` directly, as anticipated. Moreover, it has used a trick to combine the sum and the loading of `rax` into a single instruction. The `lea` instruction stands for Load Effective Address and its general form is as follows:

`lea register, [expression]`

It computes *expression* just like the instructions that access memory (Section A.2.1.1), but then stores the result of the expression in *register*, without accessing memory.

The optimization is even more dramatic in the case of `main` and it is the effect of many more optimization passes, besides register allocation: the compiler has inlined `foo`, noticed that it was computing a constant value, and then it has replaced the entire call with this value. ■

If register allocation is not enabled (as is the case when we compile without any optimization options), it is very unlikely that the compiler will use any of the non scratch registers, and therefore we will almost never observe pushes after the prologue and pops before the epilogue.

A.4 The ELF format

The Executable and Linkable Format (ELF [14]) is a file format that was introduced in System V and is now used in by all Linux distributions. ELF is a generic, extensible format which is applied to specific processors and operating systems via supplementary specifications. In our case, this specification is included in the AMD64 System V ABI [59].

As its name implies, the ELF format can be used for both executables (type EXEC) and object files (type REL). It can also be used for other purposes, such as dynamic libraries (type DYN; see Appendix B) and “core dumps” (type CORE).

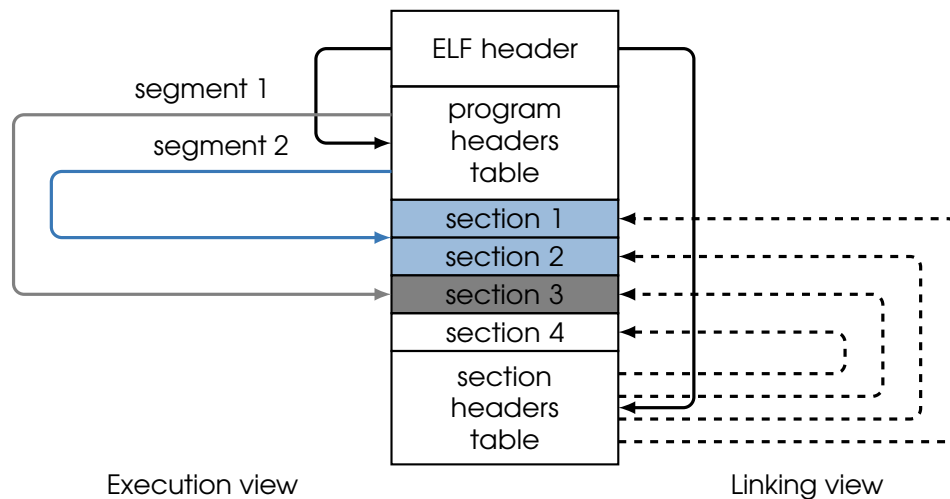


Figure A.4 – The ELF file format

The same ELF file can be viewed in two different ways, one for execution and one for linking (see Figure A.4).

- In the *linking view*, the file contains a number of non-overlapping *sections*. Some sections contain program parts, such as code or data, while others contain data structures needed by the linker, such as symbol tables or relocations.
- In the *execution view*, the file contains a number of *segments*. Segments consists of a sequence of contiguous sections and contain bytes that should be loaded into memory to create the initial state of a process. Some segments may be nested within other segments.

The ELF file contains a *section header table*, with a *section header* for each section, and a *program header table*, with a *program header* for each segment. Each header contains the starting offset and the size of the corresponding section or segment in the file, among other information. One of this tables may be absent if the corresponding view is not needed. For instance, executables may lack the section header table, while object files typically lack the program header table. These tables can have a variable number of entries and can be placed almost anywhere in the file. However, each ELF file begins with a standard *ELF header* containing all the necessary information to locate the tables. Figure A.4 shows an ELF file with four sections and two segments. Segment 1 contains sections 1 and 2, while segment 2 contains only section 3. Section 4 does not belong to any segment.

Linux used to be very lax about verifying the integrity of ELF header fields. For example, you could place the program headers table within the ELF header itself, as demonstrated in Brian Raiter’s famous blog post [82]. However, trying to reproduce the same effect on a modern 64-bit kernel yields less impressive results [72].

The `objdump` tool, that we have already used to disassemble the code contained in ELF files, can also be used to inspect other parts of the file. The preferred tool for doing this, however, is `readelf`.

■ **Example A.10** We can examine the ELF header of program `foo` of Example A.6 with:

```
$ readelf -h foo
```

The output should be something like this:

ELF Header:

```

Magic:  7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00
Class:                                     ELF64
Data:                                       2's complement, little endian
Version:                                  1 (current)
OS/ABI:                                   UNIX - System V
ABI Version:                              0
Type:                                       EXEC (Executable file)
Machine:                                  Advanced Micro Devices X86-64
Version:                                  0x1
Entry point address:                       0x401020
Start of program headers:                  64 (bytes into file)
Start of section headers:                 10536 (bytes into file)
Flags:                                     0x0
Size of this header:                       64 (bytes)
Size of program headers:                   56 (bytes)
Number of program headers:                 13
Size of section headers:                   64 (bytes)
Number of section headers:                 29
Section header string table index: 28

```

The type of this file is EXEC, meaning that the file contains a program ready to be loaded into a process for execution. Note the “Start of program headers” and “Start of section headers” fields. A zero in these fields signals that the corresponding table is missing. This file contains both tables, even if only the program headers are needed for an EXEC file. The “Number of program headers” and “Number of section headers” fields have the obvious meaning, while “Size of program headers” and “Size of section headers” may need some explanation. These are the sizes of each entry in the respective table. The sizes are stored in the ELF header to allow for the addition of fields to the entries. In this way, tools that do not know about these extensions can still be able to find each entry in the table and decode the fields they know. ■

Thanks to the name, the magic bytes (aka *magic numbers*) might seem strange or even esoteric. However, they are not. Magic bytes are simply a constant that informs applications and sometimes the kernel what they are dealing with [104]. This constant is usually placed at the beginning of a file and chosen so that it is unlikely to occur by chance or be confused with other magic bytes. Unix systems are essentially forced to use this convention, since files are untyped; their use is documented since v1 (1971). For ELF files, the magic bytes 2–4 (counting from 1) encode “ELF” in ASCII. A list of magic bytes can be found in reference [118].

A.4.1 Sections

The sections organize the file contents in a way that is useful for linkers. Each section contains either parts of the program, or some ancillary information.

Each section header describes a single section and contains the following fields:

- Name** The symbolic name of the section. Some names are predefined, but users can define their own sections with arbitrary names.
- Type** The type of the section. If the type is PROGBITS, the contents are up to the program and

not formalized by the specification.

Address If the section should be loaded in the virtual memory of a process (flag A set in Flg), this field contains its virtual address, otherwise it contains zero. Note that this field may also contain zero because the virtual address is not yet known.

Off (Offset) The offset of the first byte of the section in the file.

Size	The size of the section in bytes.
-------------	-----------------------------------

ES (Entry Size) Meaningful only for sections that contain a table of some sort. In that case, ES is the size of each entry in the table.

Flg (Flags) Miscellaneous single-bit properties of the section. The most important ones are:
A: sections that must be “allocated”, meaning they occupy space in the process’s virtual memory; X: sections that contain executable code; W: sections that must be writable when loaded in memory.

Lk (Link) The index of another section, whose meaning depends on the type of this section.

Inf (Information) Additional information whose meaning depends on the sections's type.

AI (Alignment) Required alignment. The linker must assign this section an address that is a multiple of AI. A 0 in this field is equivalent to 1.

Intra-file references to sections use the section's index in the section headers table. See, for example the Lk field above. The section header table must begin with NULL section at index 0; therefore, index 0 can be used as invalid reference.

Rather than replicating information found in the specification, we will examine some examples.

■ **Example A.11** For the purposes of the examples in this section, we will compile the file `foo2.c`, which contains the following code:

```
1 char A1[200] = "initial values";
2 const char A2[400] = "constant value";
3 char A3[600]; /* not initialized */
4 extern char A4[]; /* undefined */
5 int main()
6 {
7     A3[0] = A1[0] + A2[0] + A4[0];
8     return A3[0];
9 }
```

foo2.c

The program declares four global arrays: A1, which is initialized; A2, which is constant; A3, which is not explicitly initialized and will therefore be initialized with zeros; and A4 which, unlike the other three, is only declared and must be defined in another file.

Since sections are mostly intended for object files, we first create one by compiling `foo2.c` without invoking the linker:

```
$ gcc -c -mno-red-zone -no-pie -zexecstack \
-fcf-protection=none foo2.c
```

This creates file `f002.o`. Its ELF header is as follows:

ELF Header:

```
Magic:  7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
```

Class: ELF64

Data: 2's complement, little endian

Idx	Name	Type	Off	Size	ES	Flg	Lk	Inf	Al
(0)		NULL							
(1)	.text	PROGBITS	0x0040	0x002f	00	AX	0	0	1
(2)	.rela.text	RELA	0x0460	0x0060	18	I	11	1	8
(3)	.data	PROGBITS	0x0080	0x00c8	00	WA	0	0	32
(4)	.bss	NOBITS	0x0160	0x0258	00	WA	0	0	32
(5)	.rodata	PROGBITS	0x0160	0x0190	00	A	0	0	32
(6)	.comment	PROGBITS	0x02f0	0x0027	01	MS	0	0	1
(7)	.note.GNU-stack	PROGBITS	0x0317	0x0000	00		0	0	1
(8)	.note.gnu.property	NOTE	0x0318	0x0030	00	A	0	0	8
(9)	.eh_frame	PROGBITS	0x0348	0x0038	00	A	0	0	8
(10)	.rela.eh_frame	RELA	0x04c0	0x0018	18	I	11	9	8
(11)	.symtab	SYMTAB	0x0380	0x00c0	18		12	3	8
(12)	.strtab	STRTAB	0x0440	0x0019	00		0	0	1
(13)	.shstrtab	STRTAB	0x04d8	0x0074	00		0	0	1

Table A.1: Section headers table from Example A.11

```

Version:                1 (current)
OS/ABI:                 UNIX - System V
ABI Version:            0
Type:                   REL (Relocatable file)
Machine:                Advanced Micro Devices X86-64
Version:                0x1
Entry point address:    0x0
Start of program headers: 0 (bytes into file)
Start of section headers: 1360 (bytes into file)
Flags:                  0x0
Size of this header:    64 (bytes)
Size of program headers: 0 (bytes)
Number of program headers: 0
Size of section headers: 64 (bytes)
Number of section headers: 14
Section header string table index: 13

```

Note the REL type. There are no program headers because the file is not ready for execution.

We can inspect the section headers with

```
$ readelf -WS foo2.o
```

Table A.1 contains the output, reformatted for readability. Each line is a section header. We have removed the Address column, since the sections of an object file do not have addresses yet and all of those fields show up as zero. The Idx column is the index of the section in the section header table, not a field of the section header itself. We have included it because sections are often referred by their index.

The PROGBITS sections are the most recognizable.

- .text (index 1) contains the machine instructions. Its Flg field contains A (the section must be allocated in the process memory) and X (its contents must be executable).

- `.data` (index 3) contains the global, initialized data variables. Array A1 is allocated here, and the section contains its initial value.
- `.rodata` (index 5) contains global constants. Array A2 is allocated here.

The `.bss` section (index 4), with type NOBITS, is special because it occupies no space in the file, even if its Size field is non-zero. Since it is marked with an A, it will occupy Size bytes of process memory, and the loader must initialize it with zero. This device avoids occupying space in the file for variables that must be initialized to zero, such as uninitialized global variables in C, like our A3 array.

The names `.text`, `.data`, `.bss`, etc., predate ELF and originate from early Unix assemblers. The `.bss` name stands for “Block Started by Symbol”, an assembly pseudo-operation that Ken Thompson was probably familiar with through its use in the FORTRAN Assembly Program (FAP) for the IBM 7090. This pseudo-operation “reserve[s] an area of memory within a program for data storage or working space” [2] and simultaneously defines a symbol for its first address. In our case, the symbol is “`.bss`”.

The remaining PROGBITS sections have special uses. We will learn about `.note.GNU-stack` (index 7) in Section 9.1.4. The `.eh_frame` section (index 9) contains information about stack unwinding.

None of these sections use the ES, Lk or Inf fields. These fields are used by some of the non-PROGBITS sections that will be examined below. ■

A.4.1.1 String tables

The ELF format contains several strings, including the names of the sections and symbols. These strings are used in tables such as the section header table and the relocation tables (Section A.4.1.3). This creates an annoying problem: tables are effective only if all entries are the same size, but strings have variable lengths. Setting a maximum length for strings is too limiting if it is set low and wastes a lot of space if it is set high. The ELF format solves this problem by storing all zero-terminated strings in a dedicated section with type STRTAB, one after the other. String fields in tables only contain fixed-width offsets to the start of the string in a STRTAB section.

R This refers only to the strings defined by the ELF format itself and not to the strings defined in our program, such as “initial values” in file `foo2.c`, which will be allocated in the appropriate PROGBITS section like any other program data.

Consider for example the section headers table. The Name field is actually only an offset in the `.shstrtab` section (Section Headers String Table). The index of the section that contains the section names is written in the “Section header string table index” field of the ELF file.

■ **Example A.12** Our `foo2.o` file contains two STRTAB sections (see Table A.1): `.strtab` (index 12) and `.shstrtab` (index 13). The latter contains only the section names, while the first one contains all other strings (mostly symbol names). They are kept in separate sections because the user may decide to discard the (typically large) `.strtab` section from the final executable, but still keep the (small) `.shstrtab` section to support the section headers table.

We can ask `readelf` to show us a hexdump of any section, for example:

```
$ readelf -x .shstrtab foo2.o
```

The output of the command, reformatted and colored for readability, is:

0x00000000	002e7379	6d746162	002e7374	72746162	..symtab..strtab
0x00000010	002e7368	73747274	6162002e	72656c61	..shstrtab..rela
0x00000020	2e746578	74002e64	61746100	2e627373	.text..data..bss
0x00000030	002e726f	64617461	002e636f	6d6d656e	..rodata..commen
0x00000040	74002e6e	6f74652e	474e552d	73746163	t..note.GNU-stac
0x00000050	6b002e6e	6f74652e	676e752e	70726f70	k..note.gnu.prop
0x00000060	65727479	002e7265	6c612e65	685f6672	erty..rela.eh_fr
0x00000070	616d6500				ame.

The leftmost column shows the offset within the section. The central part shows the bytes stored at the corresponding offsets. These bytes are organized in 16-byte rows, ordered from left to right, in the same order as the “mirrored” order defined in Section A.1.2 above. Different colors highlight the bytes of the strings. The part on the right shows the same lines, but interpreted as ASCII strings. Each string is colored the same color as its corresponding bytes. The `readelf` command, like many others, renders unprintable bytes as dots. This can be confusing in this case, since there are also many actual dots (ASCII 2e). Note that the first byte of a STRTAB section is always a null byte.

In addition to the generic hexdump, which works for any section, we can ask `readelf` to print a STRTAB section as a more useful offset-string table with

```
$ readelf -p .shstrtab foo2.o
```

The output is as follows:

String dump of section '.shstrtab':

```
[ 1] .symtab
[ 9] .strtab
[11] .shstrtab
[1b] .rela.text
[26] .data
[2c] .bss
[31] .rodata
[39] .comment
[42] .note.GNU-stack
[52] .note.gnu.property
[65] .rela.eh_frame
```

The offsets on the left are the ones that are used to reference the strings in the section headers.

User-defined sections can also contain strings in the same format as a STRTAB, if they are marked with the S flag in the Flg field. In the `foo2.o` file in our example, this is the case for the `.comment` section. The “`readelf -p`” command can be used to read the strings in these sections. On the machine where `foo2.o` was built, the `.comment` section contains a single string with the version of `gcc` and the name of `gcc`’s Ubuntu package. The section is marked with the M (Merge) flag, which has the following effect: at link time, the linker looks for other sections with the same name and flags in all the other object files and merges them. This eliminates all duplicate strings, which will likely be effective in this case since the `gcc` version will be the same for all the files.

A.4.1.2 Symbol tables

The labels that we define in the assembly files become *symbols* in object files. A symbol is a name that stands for a number, called the symbol's *value*. Most commonly, the value of a symbol is an address, such as that of a variable or function.

In ELF, every symbol is relative to a section and refers to the address of a byte within that section. An ELF section with type SYMTAB is a *symbol table* that maps every symbol to the index of its section and to its value, plus some other ancillary information.

R Some special section indexes are used for special cases that do not fit into this scheme. The most important special index is UND, for symbols that are not defined in the current file. Another special index is ABS, for symbols that are just symbolic names for “absolute” values that are not associated with a particular section.

The address of a byte referred to by a symbol is not known until the linker has created the full program, ready to be loaded and executed. Until then, the symbol table only provides the *offset* of the byte within the section.

Once the linker has created the final program image, the symbols are no longer needed and the symbol table can be discarded. However, the linker typically keeps the symbol table in the executable file, since it may be useful for documentation and debugging. You can remove it using the `strip` command.

■ **Example A.13** We can examine the symbol table of our `foo2.o` file with

```
$ readelf -s foo2.o
```

We obtain the following output

Symbol table '.symtab' contains 8 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	0000000000000000	0	FILE	LOCAL	DEFAULT	ABS	foo2.c
2:	0000000000000000	0	SECTION	LOCAL	DEFAULT	1	.text
3:	0000000000000000	200	OBJECT	GLOBAL	DEFAULT	3	A1
4:	0000000000000000	400	OBJECT	GLOBAL	DEFAULT	5	A2
5:	0000000000000000	600	OBJECT	GLOBAL	DEFAULT	4	A3
6:	0000000000000000	47	FUNC	GLOBAL	DEFAULT	1	main
7:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	A4

The Ndx column indicates the index of the section in which the symbol is defined. Since this is a REL file, the Value field contains the offset of the symbol in the section. Note that there is a symbol for the `main` function, as well one for each of our global variables (A1, A2, A3 and A4). The `main` symbol is defined at the start (offset 0) of section 1. According to Table A.1, this is the `.text` section. A1, A2, and A3 are at the start of the `.data`, `.rodata`, and `.bss` sections, respectively. A4 is undefined (special section UND). There is also a symbol for `.text`, which is defined at the beginning of section 1 (which is indeed `.text`). The linker uses this symbol to assign an address to the section itself. Finally, there is an ABS symbol containing the name of the source file.

Note that symbols have a Type and a Size. We can recognize the size of the arrays we declared, but these fields are rarely used. The Bind(ing) field is important because when the linker searches object files to find the definition of an undefined symbol, it skips the LOCAL symbols. The Vis(ibility) field will be discussed briefly in Section B.6. If it contains DEFAULT, we can ignore it.

As usual, the Name fields contain only offsets into a STRTAB section. In Linux, the index of the STRTAB section that must be used for this purpose is stored in the Lk field of the `.symtab` section header. According to Table A.1, `.symtab` (index 11) has `Lk = 12`, which is `.strtab`. The same Table also shows that each symbol table entry is 24 bytes long (`ES = 0x18`). The Inf field, which is OS-specific in SYMTAB sections, is the index of the first GLOBAL symbol in the table. This is an optimization that helps the linker skip all LOCAL symbols during lookup.

The first symbol table entry is always filled with zeros, to represent an invalid symbol. This symbol also has an empty name, since a zero in the Name field points to the first byte of the STRTAB section, which is always null.

Symbols can also be examined using the ancient `nm` tool (`nm` is short for “names”):

```
$ nm foo2.o
```

Its output is simpler and therefore more readable. For each symbol, it shows its value, a letter code and the symbol's name. The letter code can be “u” for undefined, “t” for text, “d” for data, “r” for read-only data, and “b” for bss. The letter is lowercase if the symbol is local, otherwise it is global. The output is sorted by name by default. Pass “-n” if you want it sorted numerically by value. ■

A.4.1.3 Relocations

When the assembler does not know the value of a symbol, it uses a temporary value, usually 0, and then creates a *relocation instruction* for the linker. This instruction tells the linker to perform some simple calculation on the symbol's value and then modify the bytes produced by the assembler.

In the ELF format, relocations are organized in tables stored in REL or RELA sections. AMD64 only uses RELA relocations, which stand for “Relocation with Addend”, since these relocations use both a symbol and an addend.

Typically, REL relocations also need an addend; however, this addend is stored in place, i.e., in the bytes that need to be patched. This is not always possible in AMD64 because the addend generally needs to be a 64-bit constant, and most AMD64 instructions only reserve four bytes for addresses (see Section A.2.1.1). This is why there is a need for RELA relocations that store the 64-bit addend in the relocation entry itself.

Each RELA section contains relocation instructions for one other section, whose index is stored in the Inf field of the section's header.

The I flag in the section header of RELA sections indicates that the Info flag contains the index of another section (see Table A.1). Tools such as `objcopy` use this information to find all references to a section. For instance, “`objcopy -R .eh_frame`” removes the `.eh_frame` section *and* the `.rela.eh_frame` section that references it.

Each relocation must specify three pieces of information: the *place* where the relocation must be performed, expressed as an offset inside the affected section; the *type* of the operation to be performed, drawn from a small set of processor-specific operations; the *parameters* of the operation, involving a symbol and a constant addend (and sometimes some implicit operands). The symbol is referenced using its index in a SYMTAB section. Each RELA section uses a single SYMTAB section, the index of which is found in the Lk field of the section's header.

■ **Example A.14** Table A.1 shows two RELA sections at indexes 2 and 10. Both sections use the

SYMTAB section at index 11 (Lk = 11), which is `.symtab`. The RELA section at index 2 is relative to section 1 (Inf = 1), or section `.text`, and the one at index 10 is relative to section 9 (Inf = 9), or section `.eh_frame`. Note that, by convention, the RELA section is named after the section to which the relocations apply: section 2 is called `.rela.text` and section 10 is called `.rela.eh_frame`.

We can examine all the relocation instructions contained in `foo2.o` using the command

```
$ readelf -r foo2.o
```

The output shows the relocations for `.text` and `.eh_frame`. Focusing only on the `.text` relocations, they are as follows:

Offset	Info	Type	Symbol	Addend
0x007	0x0003000000002	R_X86_64_PC32	A1	-4
0x017	0x0007000000002	R_X86_64_PC32	A4	-4
0x01f	0x0005000000002	R_X86_64_PC32	A3	-4
0x026	0x0005000000002	R_X86_64_PC32	A3	-4

We omitted the “Symbol value” field, which was always zero. Additionally, we split the “Sym. name + addend” column into Symbol and Addend.

The 64-bit Info field encodes the information that we can see decoded in the two columns to its right. In particular, the four most significant bytes contain the symbol index and the four least significant bytes encode operation type. For example, the first entry contains a 3 in the four most significant bytes of Info field (note that the two most significant bytes are not shown), which corresponds to symbol A1, as confirmed by the output of “`readelf -s`” in Example A.13. The type is 2, which corresponds to `R_X86_64_PC32`.

The `R_X86_64_PC32` operation, used by all four relocation entries in this example, instructs the linker to add the Symbol and the Addend, subtract the address of the place, and overwrite the place with the result. The place is assumed to span four bytes and the result is written as a 32-bit 2’s complement integer, in little-endian format.

For instance, the first entry works on a place that is at offset 7 from the beginning of the `.text` section. The linker knows the address of the place (call it p) and the value of the symbol (i.e., the address of A1; call it a). It computes $a - 4 - p$ and then overwrites the place with the result. The purpose of this computation becomes clear when we examine the place in the `.text` section that is being patched. The disassembly around offset 7 contains the following:

```
4: 0f b6 05 00 00 00 00      movzx eax,BYTE PTR [rip+0]
```

The compiler generated an instruction that accesses A1 with `rip`-relative addressing. However, the assembler could not compute the required offset (see Section A.2.1.1), because it did not know where A1 and the instruction would end up in the final executable. Even if both the instruction and the variable come from the same source file, they exist in different sections (i.e., `.text` vs. `.data`), and the linker allocates each section independently. The unknown offset should occupy the four colored bytes in the disassembly above. The assembler used a dummy value of 0 and created the relocation instruction for the linker to patch those bytes once the addresses are known.

A small technical point regarding how this relocation is computed is that the linker does not disassemble the instructions and does not know where any instruction begins or ends. Therefore, it cannot actually compute the offset between an *instruction* and A1. The `R_X86_64_PC32` relocation actually uses the address p of the *place*, which is the address of the first colored byte in the disassembly above. However, the assembler knows that the address of the next instruction is $i = p + 4$. This is

why the relocation contains a -4 addend; the linker will then compute $a - 4 - p = a - (p + 4) = a - i$, which is the required `rip`-relative offset. ■

A.4.2 Segments

The “segments” view of the ELF file is used to load a program into a process’s virtual memory. The `execve()` system call only examines the file’s segments to determine what needs to be loaded.

Each segment is a sequence of bytes in the file. In most cases, they are used to initialize the corresponding bytes in the process’s memory.

Each program header describes a single segment, and contains the following fields:

Type	Each segment has a type that determines its use. The most important type is <code>LOAD</code> , which describes segments that must be loaded into process memory. Other types are typically used for segments that are contained within other <code>LOAD</code> segments, and provide information about the meaning of some of these bytes (we will look at some of these in the examples).
Offset	The offset in the file of the first byte of the segment.
VirtAddr	The virtual address where the first byte of the segment should be loaded into process memory.
PhysAddr	Unused in Linux. It always contains the same address as <code>VirtAddr</code> .
FileSiz	The number of bytes of the segment that are stored in the file, starting from <code>Offset</code> .
MemSiz	The number of bytes that the segment will occupy in memory, starting from <code>VirtAddr</code> . This size may be larger than <code>FileSiz</code> . In that case the loader will have to zero out the additional bytes.
Flg	Flags. Any combination of <code>R</code> (read), <code>W</code> (write) and <code>E</code> (execute). These specify the access permissions that all the bytes of the segment should have when loaded into memory.
Align	The alignment of the segment. The <code>VirtAddr</code> and the <code>Offset</code> must be congruent modulo <code>Align</code> . A zero in this field means that no particular alignment is required.

■ **Example A.15** We can create an executable from `foo2.o`, the file from Example A.11, by providing a definition for `A4` and then calling the linker. We define `A4` in a different source file, `a4.c`, as follows:

```
1 char A4[100];
```

`a4.c`

We compile `a4.c` to obtain `a4.o` and then link everything together to create the `foo2` executable:

```
$ gcc -o foo2 -no-pie -znoelro -zexecstack foo2.o a4.o
```

Note that `gcc` automatically adds several other object files and a few libraries. In particular, it adds a file containing the `_start` symbol, which becomes the entry point of our program. It also adds the `C` library. The linker uses the input section information from all these files to build the output sections of the final executable. In the process, it concatenates together all the sections with the same name and chooses a base address for each resulting section. This fixes the address of every symbol and instruction, enabling the linker to apply all the relocations. Finally, the linker assembles the relocated sections into segments and produces the program headers table. The linker discards the relocation sections from the output file, but typically keeps the symbol tables, which now contain the final address of each symbol, for documentation purposes. It also keeps the string tables that contain the names of the symbols, and the section header table needed to locate the symbol table. The section header table contains the headers for the output sections created by the linker and documents how each segment has

SegN	Type	Offset	VirtAddr	FileSiz	MemSiz	Flg	Align
(0)	PHDR	0x0040	0x400040	0x2d8	0x2d8	R	0x8
(1)	INTERP	0x035c	0x40035c	0x01c	0x01c	R	0x1
(2)	LOAD	0x0000	0x400000	0x470	0x470	R	0x1000
(3)	LOAD	0x1000	0x401000	0x145	0x145	RE	0x1000
(4)	LOAD	0x2000	0x402000	0x2cc	0x2cc	R	0x1000
(5)	LOAD	0x22d0	0x4032d0	0x2b8	0x5b8	RW	0x1000
(6)	DYNAMIC	0x22e0	0x4032e0	0x190	0x190	RW	0x8
(7)	NOTE	0x0318	0x400318	0x020	0x020	R	0x8
(8)	NOTE	0x0338	0x400338	0x024	0x024	R	0x4
(9)	NOTE	0x223c	0x40223c	0x090	0x090	R	0x4
(10)	GNU_PROPERTY	0x0318	0x400318	0x020	0x020	R	0x8
(11)	GNU_EH_FRAME	0x21b0	0x4021b0	0x024	0x024	R	0x4
(12)	GNU_STACK	0x0000	0x000000	0x000	0x000	RWE	0x10

Table A.2: The program headers of file `foo2` of Example A.15

been composed.

We can inspect the program headers of `foo2` using the following command:

```
$ readelf -Wl foo
```

Table A.2 shows the output of the command, reformatted for readability. Each line of the table represents a program header. The unused “PhysAddr” column has been omitted, and a new column has been added with the index of the segment. Note that, since the section headers table is still available, the command also shows which sections are contained in each segment. For example, the command shows the following for segment 3:

```
03      .init .text .fini
```

This indicates that segment 3 contains the `.init`, `.text` and `.fini` sections. The `.text` section contains the `.text` section of `foo2.o`, as well as other code coming from `.text` sections in files added by `gcc`. The `.init` and `.fini` sections also come from these additional files and contain code that is executed when the program starts and ends.

The most important segments are the four with `LOAD` type, at indexes 2–5. For instance, the `LOAD` segment at index 2 spans bytes at offsets $[\text{Offset}, \text{FileSz}] = [0x0000, 0x0470)$. It should be loaded at address $\text{VirtAddr} = 0x400000$ and marked as readable ($\text{Flg} = \text{R}$). The `MemSiz` is equal to the `FileSiz`, so no additional bytes need to be initialized to zero. The `LOAD` segments at indexes 3 and 4 can be interpreted similarly. The `LOAD` segment at index 5 is an instance where `FileSiz` (`0x2b8`) differs from `MemSiz` (`0x5b8`). The extra bytes come from the `.bss` section, that has been placed at the end of this segment. The loader must load `0x2b8` bytes, and then initialize an additional 656 bytes to zero (`0x5b8 - 0x2b8`).

Most of the non-`LOAD` segments are used to specify the contents of subparts of a `LOAD` segment. For example, the `PHDR` segment describes the program header table itself. This segment is contained within the first `LOAD` segment. Another example is the `DYNAMIC` segment, which is used by the dynamic linker (see Appendix B) and is contained in the fourth `LOAD` segment. These sub-segments are loaded into memory as part of the larger `LOAD` segments; their headers can then be used to retrieve them inside the containing segments.

- R** Note that the sections can be used for the same purpose and that some of these sub-segments coincide with a single section. However, while the program headers table is mandatory for an executable file, the section headers table is not. Therefore, there is no guarantee that the section information will always be available.

The INTERP segment relates to dynamic libraries and is described in Appendix B. The NOTE and GNU_PROPERTY segments contain various metadata. GNU_PROPERTY is an alias for the first NOTE segment, which it completely overlaps. You can read all NOTE sections with “`readelf -n foo2`”. The GNU_EH_FRAME segment is used for stack unwinding at runtime and by debuggers and other tools that need to display stack backtraces. GNU_STACK is an example of an empty segment. Its purpose is discussed in Section 9.1.4.

The left part of Figure A.5 shows how the segments of Table A.2 are laid out in the file. The LOAD segments in the file are colored. Note that the third and fourth LOAD segments almost touch.

As the image shows, the first LOAD segment contains mostly metadata. The second LOAD segment is the only one with execution permissions and contains the program code. The third LOAD segment contains read-only data, including data defined in the program, such as the constant A2 array, as well as data added by the compiler, such as the GNU_EH_FRAME sub-segment. The fourth LOAD segment contains read-write data, including the A1 and A4 arrays, as well as data added by the linker, such as the DYNAMIC segment.

Note that most of the file is unused due to the LOAD segments’ large alignment constraints. In particular, the white space between the first two LOAD segments, and the other one between the second and third, is wasted space that typically contains zeros.

- R** These alignment requirements do not come from the original sections, which had much smaller values (see the A1 column in Table A.1). They originate from the configuration of the linker itself. For GNU ld, this configuration is contained in a *linker script* installed in a standard location within the file system.

There are also parts of the file that are used but are not covered by segments. For example, the section header table at the end of the file is excluded since it is not needed at runtime and does not need to be loaded into memory. Almost all of the space after the fourth LOAD segment contains information, such as the symbol and the string tables, that the linker has left as documentation, but that is not actually needed to run the program. To save space, this information can be removed from the file with

```
$ strip foo2
```

This command preserves the section header table, however. To get rid of that as well, use:

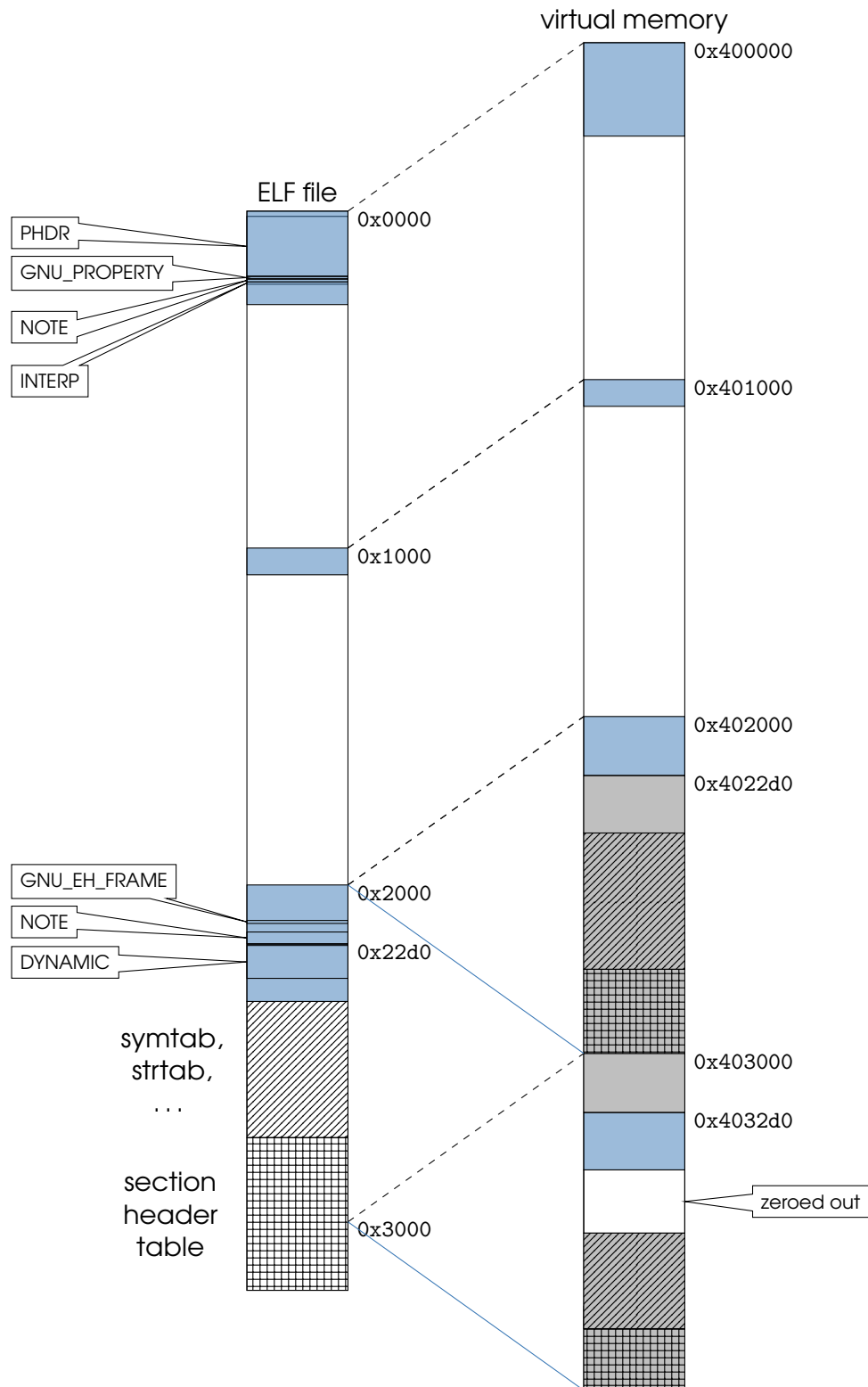
```
$ strip --strip-section-headers foo2
```

After executing this command, the `foo2` file will end just after the end of the fourth LOAD segment. ■

Note that the loader can load a larger portion of the file containing a segment as long as all constraints are met. In our ABI, we know that memory mappings and permissions have a page-sized granularity. In fact, the kernel loads an integral number of pages containing each segment.

- R** The “load” is actually implemented by creating memory mappings from virtual pages to blocks of the file. This process is essentially identical to what can be achieved using the `mmap()` system call. Actual loading from the file to memory occurs on demand, driven by page faults.

Additionally, due to limitations in the hardware, the kernel may have to grant a larger set of permissions than what it finds in the Flg fields. For example, in our MMU, writable implies readable. Most importantly, we assume that readable is equivalent to executable.

**Figure A.5** – Segments and mappings from Example A.16

■ **Example A.16** Consider the segments in Table A.2 again. Figure A.5 shows how the segments are loaded into virtual memory during an `execve()`. Dashed lines represent read-only mappings, while continuous colored lines represent read-write mappings.

To load the first LOAD segment, the kernel creates a read-only mapping from the virtual address range `[0x400000, 0x401000)` to the file offsets `[0x0000, 0x1000)`. This page-sized mapping includes the actual bytes of the segment at offsets `[0x0000, 0x0470)`, as well as other meaningless bytes.

The kernel creates a second mapping from `[0x401000, 0x402000)` to load the second segment. This segment should be marked as readable and executable, but we will assume that the kernel marks it only as readable.

Something interesting happens in the third and fourth segments. To load the third segment, the kernel creates a read-only mapping from `[0x402000, 0x403000)` to `[0x2000, 0x3000)`. Note that this range also includes the fourth segment, which will therefore be mapped as well, but not at its intended address (see the first gray region from above in the virtual memory diagram on the right). This range also includes the upper half of the section header table, which will be mapped as well. The same occurs with the symbol table, string table, etc., which are stored between the fourth segment and the section header table.

For the fourth LOAD segment, the kernel creates a mapping from the virtual address range `[0x403000, 0x404000)` to the file offset range `[0x2000, 0x3000)`. In other words, it also maps bytes that precede the segment in the file. In this case, these bytes include the third LOAD segment. Thus, besides the intended copy at address `0x402000` a second copy of the third LOAD segment will be loaded at address `0x403000`. The upper half of the program headers table, the symbol table, etc., will also be mapped again. These unintended copies are harmless, but others may not be. In particular the code, which is contained in the second LOAD segment, has been isolated into its own page to ensure that no data byte is marked as executable. The importance of this will become clear in Chapter 9.

While loading the fourth segment, the kernel must also initialize the additional bytes up to the requested `MemSiz` to zero. Note that this overwrites part of the second copy of the symbol table.

When we use the `strip` command to strip the `foo2` binary, as suggested in Example A.15, the third and fourth virtual memory pages extend beyond the end of the file. The kernel allows this, and the missing bytes of both pages are initialized with zero. ■