Software

Compilation

A **compiler**:

- reads source code,
- forms chunks of
  - data (constants, initial values for global variables)
  - executable machine code (functions)
- associates a *symbol* to each chunk (variable or function name)
- writes all into an “object” (“.o”) file (format: ELF, COFF, Mach-O)

The compiler leaves blank all *references* to symbols
(incl. external symbols like global variables and global functions)

Example:

```c
#include <stdio.h>

int the_number = -1;

int main()
{
    scanf("%d", &the_number);
    return 0;
}
```

```
...
OBJECT  GLOBAL DEFAULT    the_number
...

0000000000000000 <main>:
   0:   48 83 ec 08             sub    rsp,0x8
   4:   be 00 00 00 00          mov    esi,0x0
   9:   bf 00 00 00 00          mov    edi,0x0
   e:   31 c0                   xor    eax,eax
  10:   e8 00 00 00 00          call   15 <main+0x15>
  15:   31 c0                   xor    eax,eax
  17:   48 83 c4 08             add    rsp,0x8
  1b:   c3                      ret
```

Linking

A **linker** reads “object” files and writes an executable file.

- it assigns a position in memory to every chunk of code and data
- it sets the value of the corresponding symbol to the address of this position
- it resolves all references to symbols:

  replaces all references with the numeric value of the corresponding position in memory

Example:

```c
#include <stdio.h>

int the_number = -1;

int main()
{
    scanf("%d", &the_number);
    return 0;
}
```

```
...
    54: 000000000040400c     4 OBJECT  GLOBAL DEFAULT   24 the_number
...
    63: 0000000000401040    28 FUNC    GLOBAL DEFAULT   14 main
...
0000000000401040 <main>:
  401040:       48 83 ec 08             sub    rsp,0x8
  401044:       be 0c 40 40 00          mov    esi,0x40400c
  401049:       bf 10 20 40 00          mov    edi,0x402010
  40104e:       31 c0                   xor    eax,eax
  401050:       e8 db ff ff ff          call   401030 <__isoc99_scanf@plt>
  401055:       31 c0                   xor    eax,eax
  401057:       48 83 c4 08             add    rsp,0x8
  40105b:       c3                      ret
```

Why a separate compilation and linking phase?

Advantages:

- Simplifies compilations

  (allows the compiler to write code using functions and variables it has not seen yet)
- Allows breaking down code into multiple files…

  that can be compiled separately
- It allows using code written/compiled by other people

  saves time, enables closed-source software

Drawbacks:

- Compiler does not know the code inside other object files
  - cannot check for mistakes based on that knowledge
  - cannot optimize code based on that knowledge

Static and dynamic linking

- Static linking is performed in order to prepare an executable (`.exe`, ...) file.
- Dynamic linking is performed every time the executable is run
  - Object files built to be dynamically linked are called
    - shared objects (`.so`, Linux and MacOS), or
    - dynamically-linked libraries (`.dll`, Windows)
  - Typically used for
    - System libraries
    - Plugins

Why dynamic linking?

- Dynamic linking allows us to use system libraries without shipping them
- It reduces the size of executables
- It helps in masking some system incompatibilities

  (e.g. run the same `.exe` on Windows 10 and 11)
- It allows updating system libraries separately

- Drawback: Dynamically-linked libraries add complexity

  (separate installation, incompatible versions, etc.)

Libraries

Libraries are collections of functions (and data) that can be used by different executables

Examples:

- `libjpeg`: read/write jpeg files
- `libssl`: cryptography
- `BLAS`: fast vector and matrix operations
- `Qt`: cross-platform GUI toolkit

Most languages have a standard library

- Distinct from the language itself, but usually necessary in any program
- The C language provides no functions.

  (All basic utilities (`strlen`, `printf`, `exit`) come from the standard library.)
- It is normally dynamically linked

Optimizing compilers

```c
int main()
{
    int r = 0;

    for (int i = 0; i < 1000000; i++)
        r = r + 2;

    return r;
}
```

```
0000000000401020 <main>:
  401020:       b8 80 84 1e 00          mov    eax,0x1e8480  # <-- 2,000,000
  401025:       c3                      ret
```

Note

“Optimal” = “best”

“Optimizing” = “going towards the best possible result”

Do not say: “I made my code more optimal”

Do say: “I optimized my code some more”

or “I made my code better”

Operating Systems

The operating system (OS) manages the computer and provides services to applications.

Components:

- The kernel handles:
  - most of the boot process (what happens upon power on)
  - memory allocation and sharing
  - input/output devices, through “drivers” (often dynamically loaded)
  - application coexistence and cooperation
- Optionally:
  - Standard libraries for some languages (C, C++, .NET, Swift, …)
  - Some additional common libraries
  - User interface (UI): command-line (CLI), graphical (GUI)
  - Some tools: CLI utilities, compilers, settings/configuration apps

Popular OSs:

- Windows
- MacOS, iOS (base OS: Darwin, kernel: XNU)
- Android (kernel: Linux)

Other current OSs:

- SteamOS, Debian, Ubuntu, Suse, Fedora, Arch, RHEL, AL2 (base OS: GNU, kernel: Linux)
- OpenWrt (base OS: BusyBox, kernel: Linux)
- FreeBSD, OPNsense, TrueNAS, pfSense (base OS & kernel: FreeBSD)
- OpenBSD

All the above except Windows are descendants from “Unix”

```c
FILE *f = fopen("my_file.txt", "r");
```

On my system:

- `fopen()` is part of the standard library
- `fopen()` calls Unix-specific `open()`, also in the standard library
  on Windows it would call `CreateFile()`
- `open()` is a wrapper for the open system call in the Linux kernel

  ```
  # open("my_file.txt", O_RDONLY);
  mov rdi, 0x402010   # pointer to "my_file.txt"
  mov rsi, 0x0        # O_RDONLY == 0
  mov rax, 2          # open is syscall #2
  syscall
  ```

- the Linux kernel uses its filesystem and SSD drivers to open the file
- it returns a file descriptor (`int`)
- `fopen()` allocates a structure with buffers and the file descriptor, returns it

Levels of abstraction

- the processor only does elementary operations (move 64-bit to/from memory)
- the kernel implements basic functionality (managing devices, reading data from a file)
- the standard library provides more, OS-independent functionality (buffering, parsing data)
- other libraries may allow even more (e.g. decompressing a video file)

Virtualized memory

Recall this example:

```c
#include <stdio.h>

int the_number = -1;

int main()
{
    scanf("%d", &the_number);
    return 0;
}
```

```
...
    54: 000000000040400c     4 OBJECT  GLOBAL DEFAULT   24 the_number
...
    63: 0000000000401040    28 FUNC    GLOBAL DEFAULT   14 main
...
0000000000401040 <main>:
  401040:       48 83 ec 08             sub    rsp,0x8
  401044:       be 0c 40 40 00          mov    esi,0x40400c
  401049:       bf 10 20 40 00          mov    edi,0x402010
  40104e:       31 c0                   xor    eax,eax
  401050:       e8 db ff ff ff          call   401030 <__isoc99_scanf@plt>
  401055:       31 c0                   xor    eax,eax
  401057:       48 83 c4 08             add    rsp,0x8
  40105b:       c3                      ret
```

Memory is virtualized

- every process sees memory as if it was alone
- every time a process accesses memory,
  the **hardware** translates the virtual address into a hardware address
- the translation uses a *page table* managed by the kernel

Page table (managed by the kernel):

page	virtual address	hardware address
#0	0 – 4095	65536 – 69631
#1	4096 – 8191	20480 – 24575
#2	8192 – 12287	4096 – 8191
…	…	…

```
# x86_64
mov eax, DWORD PTR [4100]
```

```
# AArch64
ldr w0, [4100]
```

- the processor looks up virtual address 4100 in the page table
- it finds page #1, base 4096, plus offset 4
- page #1 has hardware address 20480
- the memory access is at hardware address 20480 + 4 = 20484

Page table

- the page table itself is in memory!
- at a specific hardware address
- various techniques to make page lookup faster (it is actually a tree, with a cache)

Memory allocation

- the kernel finds free hardware addresses (unused by **any** process)
- for the virtual addresses:
  - either the process requests specific virtual addresses
  - or the kernel finds free virtual addresses (unused by **this** process)
- the kernel adds suitable entries in the page table
- the kernel returns the virtual address to the process

Virtual memory

Cons:

- slow!
- memory sharing between processes must be (initially) mediated by the kernel

Pros:

- simplifies memory management for the process
- enables process isolation (a process cannot snoop on or crash another)
- enables fast move for large chunks of memory (just update the page table)
- allows fast input/output on devices

  (non-memory devices can be mapped to virtual addresses)
- allows extending memory:
  - using storage devices (“swap”)
  - using compression
  - using overcommit

Stack

```c
void f1(void)
{
    uint64_t a, b;

    f2();
    f3();
}
```

```c
void f2(void)
{
    uint64_t c;

    f3();
}

int f3(void)
{
}
```

```
...  16  24  32  40  48  56  64  72  80  ...

...

       ^
```

```
f1(): allocate 2 x uint64_t
```

```c
void f1(void)
{
    uint64_t a, b;

    f2();
    f3();
}
```

```c
void f2(void)
{
    uint64_t c;

    f3();
}

int f3(void)
{
}
```

```
…   16   24   32   40   48   56   64   72   80   …
…    a    b
               ^
```

```
f1(): call f2
```

```c
void f1(void)
{
    uint64_t a, b;

    f2();
    f3();
}
```

```c
void f2(void)
{
    uint64_t c;

    f3();
}

int f3(void)
{
}
```

```
...  16  24  32  40  48  56  64  72  80  ...
...   a   b   R
                  ^
```

```
f2(): allocate 1 x uint64_t
```

```c
void f1(void)
{
    uint64_t a, b;

    f2();
    f3();
}
```

```c
void f2(void)
{
    uint64_t c;

    f3();
}

int f3(void)
{
}
```

...	16	24	32	40	48	56	64	72	80	...
...	a	b	R	c						
					^					

`f2(): call f3()`

```c
void f1(void)
{
    uint64_t a, b;

    f2();
    f3();
}
```

```c
void f2(void)
{
    uint64_t c;

    f3();
}

int f3(void)
{
}
```

...	16	24	32	40	48	56	64	72	80	...
...	a	b	R	c	R					
						^				

`f3(): return (to f2())`

```c
void f1(void)
{
    uint64_t a, b;

    f2();
    f3();
}
```

```c
void f2(void)
{
    uint64_t c;

    f3();
}

int f3(void)
{
}
```

…	16	24	32	40	48	56	64	72	80	…
…	a	b	R	c						
					^					

f2(): return (to f1())

```c
void f1(void)
{
    uint64_t a, b;

    f2();
    f3();
}
```

```c
void f2(void)
{
    uint64_t c;

    f3();
}

int f3(void)
{
}
```

```
...  16  24  32  40  48  56  64  72  80  ...
...   a   b
              ^
```

```
f1(): call f3()
```

```c
void f1(void)
{
    uint64_t a, b;

    f2();
    f3();
}
```

```c
void f2(void)
{
    uint64_t c;

    f3();
}

int f3(void)
{
}
```

...	16	24	32	40	48	56	64	72	80	...
...	a	b	R							
				^						

`f3(): return (to f1())`

```c
void f1(void)
{
    uint64_t a, b;

    f2();
    f3();
}
```

```c
void f2(void)
{
    uint64_t c;

    f3();
}

int f3(void)
{
}
```

...	16	24	32	40	48	56	64	72	80	...
...	a	b								
			^							

`f1(): return`

```c
void f1(void)
{
    uint64_t a, b;

    f2();
    f3();
}
```

```c
void f2(void)
{
    uint64_t c;

    f3();
}

int f3(void)
{
}
```

```
...  16  24  32  40  48  56  64  72  80  ...

...

      ^
```

Back to initial state

Stack pointer

- x86_64: rsp (by convention – rsp is a general register)
- AArch64: sp (mandatorily – sp is a special register)
- In both cases, the stack actually grows downwards
- Default stack size on Linux: 8 MB
  - theoretical max recursion depth (best case): 1,000,000

Heap

People used to refer to all memory that is not on stack as "the heap".

- Not to be confused with a heap data structure.
- The term "the heap" was more relevant when it designated a single contiguous block of virtual addresses.
- Nowadays, OSs offer more flexibility for memory allocation.