Behind the System via C

Allam Koushik Reddy

School of Computers & Information Sciences,
University of Hyderabad
C R Rao Road, Gachibowli,
Hyderabad.
25mcce10@uohyd.ac.in

Key Words

1) OS Loader
2) Runtime Environment
3) Instruction cycle
4) objdump
5) Gdb
6) Static and Dynamic Libraries
7) $code_s$ - source code (“C” code that the programmer wrote)
8) $code_p$ - preprocessed code
9) $code_a$ - assembly code after compiling “C” code
10) $code_b$ - binary code after assembling
11) $code_{fml}$ - binary after linking (this is the final executable file)

Abstract

Introduction to this article

My uncle recently recounted a compelling incident from his company's archives concerning an unexpected program failure. The system involved a section of legacy code that had operated flawlessly for years, yet it suddenly began crashing. Upon investigation, the engineering team identified the failure as a fundamental architectural issue related to dynamic memory allocation. Specifically, a function initiated an array and stored its starting memory address, which we can refer to as X. It then passed this array to a subsequent function for processing. The complication arose when the second function attempted to modify the array by adding extra elements. Since all aspects of an array must occupy a single, contiguous block of memory, the system needed more space immediately following address X. When this space was unavailable due to memory fragmentation, the operating system's memory manager was forced to allocate an entirely new, sufficiently large block at a different address, Y, copy the data, and release the original address X. The core problem was that the first function was still holding and attempting to use the now-invalid, outdated address X. This rare sequence of events, triggered only under specific, low-memory conditions, resulted in an inevitable crash when the program tried to access freed memory. This experience was a stark reminder that actual programming competence demands far more than surface-level language fluency. As my uncle emphasized, while syntax is easily learned—even provided by generative AI models—the ability to reason about system architecture, memory constraints,

and the deep operational mechanics of every statement remains a uniquely human, critical-thinking skill necessary for building truly robust and reliable software.

Quick Rules

1) Why is there a “main” function?

Any C program must have a function named “main”. Compilation will be done from top to bottom in a file, but the program is always executed from the “main” function.
> The OS Loader loads your program into memory
> While loading, it jumps into a fixed entry point address
> The entry point is the start of the runtime environment called as “C runtime (CRT)”
> If C allows you to name this entry point as you wish, then the runtime could not know which function to call. It is universally decided that the function named main should be the entry point for the program.

Don’t get Confused!
A loader is an essential component of the operating system that loads programs from secondary storage into main memory, allowing the CPU to execute them. It allocates memory, loads the program's code and data, and resolves any necessary symbolic references to libraries or other modules before transferring control to the program's entry point. This differs from the Instruction cycle (Fetch - Decode - Execute), which occurs for every step: fetch the command, decode and understand what to do, and execute it. However, the OS Loader comes into play only at the beginning.

Let's experiment to confirm that OS Loader calls CRT, and CRT calls Main for improved satisfaction.

Time to experiment!
Step 1 - create a tiny "C" program and compile with complete debug information

```
koushik@debianAKR:~/Documents$ vi exp1.c
koushik@debianAKR:~/Documents$ cat exp1.c
int main() { return 42; }
koushik@debianAKR:~/Documents$ gcc exp1.c -g
koushik@debianAKR:~/Documents$ ls
a.out  Dato  exp1.c  LAMDA
koushik@debianAKR:~/Documents$
```

Step 2 - Here, you can use both the objdump utility and gdb. The objdump utility is a command-line tool in Unix-like operating systems (part of the **GNU Binutils**) used for displaying various information about **object files**, executables, and libraries. It acts like a high-powered microscope for binaries, allowing you to examine their low-level structure.
If you want to use objdump, this is the expected output.

```
koushik@debianAKR:~/Documents$ objdump -t a.out | grep start
0000000000000000       F *UND*  0000000000000000              __libc_start_main@GLIBC_2.34
0000000000004000  w      .data  0000000000000000              data_start
0000000000004000 g       .data  0000000000000000              __data_start
0000000000000000  w      *UND*  0000000000000000              __gmon_start__
0000000000001040 g     F .text  0000000000000022              _start
0000000000004010 g       .bss   0000000000000000              __bss_start
```

For deeper insight into the program, try the command $objdump -d <targetfile>
So the "_start" is a part of CRT, not your program. Learn more about "_start" in further courses.
This is the output if you want to visualize the process in GDB. You can view registers and their values, as well as assembly code in a layout.

```
(gdb) b _start
Breakpoint 1 at 0x1040
(gdb) b main
Breakpoint 2 at 0x112d: file exp1.c, line 1.
(gdb) run
Starting program: /home/koushik/Documents/a.out
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x0000555555555040 in _start ()
(gdb) n
Single stepping until exit from function _start,
which has no line number information.
0x00007ffff7de1ce0 in __libc_start_main () from /lib/x86_64-linux-gnu/libc.so.6
(gdb) n
Single stepping until exit from function __libc_start_main,
which has no line number information.

Breakpoint 2, main () at exp1.c:1
1       int main() { return 42; }
(gdb) n
0x00007ffff7de1ca8 in ?? () from /lib/x86_64-linux-gnu/libc.so.6
(gdb) n
Cannot find bounds of current function
(gdb)
```

2) How do Libraries give new abilities?

There are two kinds of libraries in C.
1) Static Libraries
2) Dynamic Libraries
They behave completely differently.

If you use a Static Library, the code is literally copied into your $code_{fnl}$. During the linking stage, the linker does this :
> Finds only the functions that you actually used
> Copies the machine $code_b$ for those functions
> pastes it into your $code_{fnl}$.

Consequences :
> Your $code_{fnl}$ becomes BIGGER
> Updating a Library does not update your program
> your program contains everything it needs
> You can run the $code_{fnl}$ without needing any system libraries, and it will have a faster startup (no loading of external libraries)

If you use a dynamic Library, the $code_b$ of that library is not copied into your $code_{fnl}$; instead, it will be loaded into memory at runtime. This is how your system works, typically by default :
> You get a $code_{fnl}$ that says "I need libxyz.so"
> But not the actual $code_b$ of the library function that you used.
At runtime, the OS Loader does this :
> Looks at your $code_{fnl}$'s "needed libraries" list
> Loads "libxyz.so" etc into main memory
> Fixes the addresses of the library's function calls (relocation)
> Then starts your program.

Consequences :
> $code_{fnl}$ is smaller
> Faster compilation
> Library updates automatically update your program
> Many programs share the same $code_b$ in memory, which saves RAM

Don't get Confused!
What is a linker? Linker is a part of a compiler. A typical compiler has a preprocessor, a compiler, an assembler, and a linker
preprocessor - It does text substitutions, file inclusion, and macro expansion.
1) It literally copies and pastes the header file content into your $code_s$ (not the library function definitions, just the function declaration)
2) Conditional compilation
3) Removes comments
4) Replaces text with other text if you have used #define

compiler - Translates the preprocessed high-level $code_p$ into architecture-specific assembly $code_a$.
Assembler - Translates the assembly $code_a$ into machine code (binary instructions) contained in a relocatable object file ($code_b$).
Linker - Combines the program's object file(s) with the necessary external library object files to create the final executable program ($code_{fnl}$).

Step	Input	Output
Preprocessing	Source $code_s$ (.c)	Preprocessed $code_p$ (.i)
Compiler	Preprocessed $code_p$ (.i)	Assembly $code_a$ (.s)
Assembler	Assembly $code_a$ (.s)	Object file / $code_b$ (.o)
Linker	Object file (.o) + libraries (.so)	Executable file / $code_{fnl}$ (.o / .out / .exe)

Time to experiment!
To view intermediate files, you can use GCC, but with flags
1) After preprocessing, you can use the -E flag to obtain the .I file after preprocessing

```
akr@KoushikReddys-MacBook-Pro experimennt2 % vi exp2.c
akr@KoushikReddys-MacBook-Pro experimennt2 % cat exp2.c
#include<stdio.h>

int main () {
        printf("hello");
        return 8;
}
akr@KoushikReddys-MacBook-Pro experimennt2 % gcc -E exp2.c -o exp2.i
akr@KoushikReddys-MacBook-Pro experimennt2 % cat exp2.i
# 1 "exp2.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 433 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "exp2.c" 2
# 1 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/stdio.h" 1 3 4
# 61 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/stdio.h" 3 4
# 1 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/_stdio.h" 1 3 4
# 69 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/_stdio.h" 3 4
# 1 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/_bounds.h" 1 3 4
# 27 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/_bounds.h" 3 4
# 1 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/sys/cdefs.h" 1 3 4
# 808 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/sys/cdefs.h" 3 4
# 1 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/sys/_symbol_aliasing.h" 1 3 4
# 809 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/sys/cdefs.h" 2 3 4
# 874 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/sys/cdefs.h" 3 4
# 1 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/sys/_posix_availability.h" 1 3 4
# 875 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/sys/cdefs.h" 2 3 4
# 992 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/sys/cdefs.h" 3 4
# 1 "/Library/Developer/CommandLineTools/usr/lib/clang/17/include/ptrcheck.h" 1 3 4
# 993 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/sys/cdefs.h" 2 3 4
# 28 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/_bounds.h" 2 3 4
# 70 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/_stdio.h" 2 3 4

# 1 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/Availability.h" 1 3 4
# 196 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/Availability.h" 3 4
# 1 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/AvailabilityVersions.h" 1 3 4
# 197 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/Availability.h" 2 3 4
# 1 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/AvailabilityInternal.h" 1 3 4
# 33 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/AvailabilityInternal.h" 3 4
# 1 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/AvailabilityVersions.h" 1 3 4
# 34 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/AvailabilityInternal.h" 2 3 4
# 198 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/Availability.h" 2 3 4
# 1 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/AvailabilityInternalLegacy.h" 1 3 4
# 34 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/AvailabilityInternalLegacy.h" 3 4
# 1 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/AvailabilityInternal.h" 1 3 4
# 35 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/AvailabilityInternalLegacy.h" 2 3 4
```

My actual code, before preprocessing, is still there, but a few hundred lines follow it.

2) You can see the assembly code after the compilation by using the flag -S

Time to experiment!

```
akr@KoushikReddys-MacBook-Pro experimennt2 % gcc -S exp2.i -o exp2.s
akr@KoushikReddys-MacBook-Pro experimennt2 % cat exp2.s
        .section        __TEXT,__text,regular,pure_instructions
        .build_version macos, 26, 0     sdk_version 26, 1
        .globl  _main                           ## -- Begin function main
        .p2align        4, 0x90
_main:                                  ## @main
        .cfi_startproc
## %bb.0:
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register %rbp
        subq    $16, %rsp
        movl    $0, -4(%rbp)
        leaq    L_.str(%rip), %rdi
        movb    $0, %al
        callq   _printf
        movl    $8, %eax
        addq    $16, %rsp
        popq    %rbp
        retq
        .cfi_endproc
                                        ## -- End function
        .section        __TEXT,__cstring,cstring_literals
L_.str:                                 ## @.str
        .asciz  "hello"

.subsections_via_symbols
akr@KoushikReddys-MacBook-Pro experimennt2 % _
```

You can compile and assemble the program using the -c flag; this is called a binary, and it cannot be viewed using a regular text editor. You have to use the objdump utility.

You will obviously obtain a regular, compiled, assembled, and linked code using a standard command.
$gcc program.c -o program_dynamic.o
And a code with a static library using this command.
$gcc program.c -static -o program_static.o
The statically linked executable (program_static) will be significantly larger than the dynamically linked one (program_dynamic).
Use objdump to see the internal differences.

3) behaviour of I/O functions in the standard input output library

This section will be truly experimental, as these implementations are not built in the C language, and a single printf function would be spread across multiple files in the library. It will be tough to obtain the code behind this library, even though you have it; understanding it is even more challenging.
We will utilize permutations and combinations to determine the number of ways we can use functions in this library, building on our previous knowledge, and gather additional data through experimentation.

Segmentation fault

Error… this section needs to be worked out