# How kernel, compiler, and C library work together

From OSDev Wiki

**Contents**

## Kernel

The kernel is the core of an operating system. In a traditional design, it is responsible for memory management, I/O, interrupt handling, and various other things. And even while some modern designs like Microkernels or Exokernels move several of these services into user space, this matters little in the scope of this document.

The kernel makes its services available through a set of system calls; how they are called and what they do exactly differs from kernel to kernel.

## C Library

> One thing up front: When you begin working on your kernel, you do not have a C library available. You have to provide everything yourself, except a few pieces provided by the compiler itself. You will also have to port an existing C library or write one yourself. See GCC Cross-Compiler, C Library, Creating a C Library.

The C library implements the standard C functions (i.e., the things declared in

<stdlib.h>, <math.h>, <stdio.h> etc.) and provides them in binary form suitable for linking with user-space applications.

In addition to standard C functions (as defined in the ISO standard), a C library might (and usually does) implement further functionality, which might or might not be defined by some standard. The standard C library says nothing about networking, for example. For Unix-like systems, the POSIX standard defines what is expected from a C library; other systems might differ fundamentally.

It should be noted that, in order to implement its functionality, the C library must call kernel functions. So, for your own OS, you can of course take a ready-made C library and just recompile it for your OS - but that requires that you tell the library how to call your kernel functions, and your kernel to actually provide those functions. The good news is that relatively few of the library's functions do use some system call (http://sourceware.org/newlib/libc.html#SEC195)

Some implementations of the standard C library include:

- GNU C library (http://www.gnu.org/software/libc/) (with info about porting the glibc (http://www.gnu.org/software/libc/manual/html_node/Porting.html) )
- newlib (http://sources.redhat.com/newlib/) (with info on the required OS functions detailed in the manual)
- uClibC (http://www.uclibc.org) (although that is highly optimized to be used with an embedded *Linux*).
- fdlibm (http://www.netlib.org/fdlibm/)
- dietlib (http://www.fefe.de/dietlibc/)

A more elaborate example is available in Library Calls or, you can use an existing C Library or create your own C Library.

## Compiler / Assembler

An Assembler takes (plaintext) source code and turns it into (binary) machine code; more precisely, it turns the source into *object* code, which contains additional information like symbol names, relocation information etc.

A compiler takes higher-level language source code, and either directly turns it into object code, or (as is the case with GCC) turns it into Assembler source code and invokes an Assembler for the final step.

The resulting object code does *not* yet contain any code for standard functions called. If you `include`d e.g. `<stdio.h>` and used `printf()`, the object code will merely contain a *reference* stating that a function named `printf()` (and taking a `const char *` and a number of unnamed arguments as parameters) must be linked to the object code in order to receive a complete executable.

Some compilers use standard library functions *internally*, which might result in object files referencing e.g. `memset()` or `memcpy()` even though you did not include the header or used a function of this name. You will have to provide an implementation of these functions to the linker, or the linking will fail. The gcc frestanding environment expects only the functions memset, memcpy, memcmp, and memmove, as well as the libgcc library. Some advanced operations (e.g. 64-bits divisions on a 32-bits system) might involve *compiler-internal* functions. For GCC, those functions are residing in libgcc. The content of this library is agnostic of what OS you use, and it won't taint your compiled kernel with licensing issues of whatever sort.

## Linker

A linker takes the object code generated by the compiler / assembler, and *links* it against the C library (and / or libgcc.a or whatever link library you provide). This can be done in two ways: static, and dynamic.

### Static Linking

When linking statically, the linker is invoked during the build process, just after the compiler / assembler run. It takes the object code, checks it for unresolved references, and checks if it can resolve these references from the available libraries. It then adds the binary code from these libraries to the executable; after this process, the executable is *complete*, i.e. when running it does not require anything but the kernel to be present.

On the downside, the executable can become quite large, and code from the libraries is duplicated over and over, both on disk and in memory.

### Dynamic Linking

When linking dynamically, the linker is invoked during the *loading* of an executable. The unresolved references in the object code are resolved against the libraries currently present in the system. This makes the on-disk executable much smaller, and allows for in-memory space-saving strategies such as *shared libraries* (see below).

On the downside, the executable becomes dependent on the presence of the libraries it references; if a system does not have those libraries, the executable cannot run.

### Shared Libraries

A popular strategy is to *share* dynamically linked libraries across multiple executables. This means that, instead of attaching the binary of the library to the executable image, the references in the executable are tweaked, so that all executables refer to the same in-memory representation of the required library.

This requires some trickery. For one, the library must either not have any *state* (static or global data) at all, or it must provide a separate *state* for each executable. This gets even trickier with multi-threaded systems, where one executable might have more than one simultaneous control flow.

Second, in a virtual memory environment, it is usually impossible to provide a library to all executables in the system at the same virtual memory address. To access library code at an arbitrary virtual address requires the library code to be *position independent* (which can be achieved e.g. by setting the -PIC command line option for the GCC compiler). This requires support of the feature by the binary format (relocation tables), and can result in slightly less efficient code on some architectures.

### ABI - Application Binary Interface

The ABI of a system defines how library function calls and kernel system calls are actually done. This includes whether parameters are passed on the stack or in registers, how function entry points are located in libraries, and other such concerns.

When using static linkage, the resulting executable depends on the kernel using the same ABI as the one the executable was built for; when using dynamic linkage, the executable depends on the libraries' ABI staying the same.

### Unresolved Symbols

The linker is the stage where you will find out about stuff that has been added without your knowledge, and which is not provided by your environment. This can include references to `alloca()`, `memcpy()`, or several others. This is usually a sign that either your toolchain or your command line options are not correctly set up for compiling your own OS kernel - or that you are using functionality that is not yet implemented in your C library / runtime environment! You will most certainly run into trouble if you are not using a cross-compiler and the libgcc library and have implementations of memcpy, memmove, memset and memcmp.

Other symbols, such as _udiv* or __builtin_saveregs, are available in libgcc. If you get errors about missing such symbols, remember that you need to link with libgcc.

## __alloca, ___main

`alloca()` is a "semi-standard" C function (from BSD?, but supported by most C implementations) that is used to allocate memory from the stack. On Windows this function is also used for stack probing. As such, `alloca()` is referenced in PE binaries built, for example, by Cygwin GCC. You can use the Cygwin GCC argument `-mno-stack-arg-probe` to suppress those references.

Another "specialty" of PE binaries is that, if you define `int main()`, a function `void __main()` is called first thing after entering `main()`. You can either define that function, or omit `main()` from your kernel code, using a different function as entry point.

This explanation of `alloca()` comes from Chris Giese, posted to alt.os.dev:

> \>> I think _alloca() is for stack-probing, which is required by Windows.
> \> What is stack probing?
> By default, Windows reserves 1 meg of virtual memory for the stack. No page of stack memory is actually allocated (committed) until the page is accessed. This is demand-allocation. The page beyond the top of the stack is the guard page. If this page is accessed, memory will be allocated for it, and the guard page moved downward by 4K (one page). Thus, the stack can grow beyond the initial 1MB. Windows will not, however, let you grow the stack by accessing discontiguous pages of memory. Going beyond the guard page causes an exception. Stack-probing code prevents this.

Some more information about stack-probing:

- http://groups.google.com/groups?&selm=702bki%24oki%241%40news.Eindhoven.NL.net
- http://groups.google.com/groups?&selm=3381B63D.6E39%40austin.finnigan.com

`alloca()` is not only used for stack-probing, but also as a sort of `malloc()`, to dynamically allocate memory for variables/buffers without the need to manually free the reserved memory (with `free()`, as you normally should).

If you search the man pages for `alloca` on a UNIX OS you'll find something like this:

> The `alloca()` function allocates space in the stack frame of the caller, and returns a pointer to the allocated block. This temporary space is automatically freed when the function from which `alloca()` is called returns.

On Windows:

> `_alloca` allocates size bytes from the program stack. The allocated space

> is automatically freed when the calling function exits (not when the allocation merely passes out of scope). Therefore, do not pass the pointer value returned by `_alloca` as an argument to free.

Note that "a stack overflow exception is generated if the space cannot be allocated", so `alloca()` might cause the stack to grow.

Unfortunately, there are all kinds of caveats with using `alloca()`. Notably, Turbo-C/C++ had the limitation that functions calling `alloca()` needed to have at least one local variable, and Linux man-page warns the following:

> The `alloca` function is machine and compiler dependent. On many systems its implementation is buggy. Its use is discouraged.
>
> On many systems `alloca` cannot be used inside the list of arguments of a function call, because the stack space reserved by `alloca` would appear on the stack in the middle of the space for the function arguments.