filename: c-writing-software-without-the-standard-library-linux-edition-20250530.txt
https://gist.github.com/tcoppex/443d1dd45f873d96260195d6431b0989

```
######################################################################
          Writing C software without the standard library
                           Linux Edition
######################################################################
```

There are many tutorials on the web that explain how to build a simple hello world in C without the libc on AMD64, but most of them stop there.

I will provide a more complete explanation that will allow you to build yourself a little framework to write more complex programs. The code will support both AMD64 and i386.

Major credits to http://betteros.org/ which got me into researching libc-free programming.

Why would you want to avoid libc?
- Your code will have no dependencies other than the compiler.
- Not including the massive header files and not linking the standard library makes compilation faster. It will be nearly instantaneous even for thousands of lines of code.
- Executables are incredibly small (the http mirror server for my gopherspace is powered by a 10kb executable).
- Easy to optimize for embedded computers that have very limited resources.
- Easy to port to other architectures as long as they are documented, without having to worry whether the libs you use support it or not.
- Above all, it exposes the inner workings of the OS, architecture and libc, which teaches you a lot and makes you more aware of what you're doing even when using high level libraries.
- It's a fun challenge!

I might not be an expert yet, but I will share my methods with you.

For now this guide is linux-only, but I will be writing a windows version when I feel like firing up a virtual machine.

```
######################################################################
                          Basic AMD64 Setup
######################################################################
```

When we learn C, we are taught that main is the first function called in a C program. In reality, main is simply a convention of the standard library.

Let's write a simple hello world and debug it.
We will compile with debug information (flag -g) as well as no optimization (-O0) to be able to see as much as possible in the debugger.

```
----------------------------------------------------------------------
$> cat > hello.c << "EOF"
#include <stdio.h>

int main(int argc, char* argv[]) {
    printf("hello\n");

    return 0;
}
EOF

$> gcc -O0 -g hello.c
$> ./a.out
hello

$> gdb a.out
(gdb) break main
(gdb) run
(gdb) backtrace
#0    main (argc=1, argv=0x7fffffffd7f8) at hello.c:6
----------------------------------------------------------------------
```

Hmm... seems like gdb is hiding stuff from us. Let's tell it that we actually care about seeing libc functions:

```
----------------------------------------------------------------------
(gdb) set backtrace past-main on
(gdb) set backtrace past-entry on
(gdb) bt
#0  main (argc=1, argv=0x7fffffffd7f8) at hello.c:6
#1  0x00007ffff7a5f630 in __libc_start_main (main=0x400556 <main>,
    argc=1, argv=0x7fffffffd7f8, init=<optimized out>,
    fini=<optimized out>, rtld_fini=<optimized out>,
    stack_end=0x7fffffffd7e8)
    at libc-start.c:289
#2  0x0000000000400489 in _start ()
```

```
-----------------------------------------------------------------------
```

That's much better! As we can see, the first function that's really
called is _start, which then calls __libc_start_main which is
clearly a standard library initialization function which then calls
main.

You can go take a look at _start __libc_start_main in glibc source
if you want, but it's not very interesting for us as it sets up a
bunch of stuff for dynamic linking and such that we will never use
since we want a static executable.

Let's recompile our hello world with optimization (-O2), without
debug information and with stripping (-s) to see how large it is:

```
-----------------------------------------------------------------------
$> gcc -s -O2 hello.c
$> wc -c a.out
6208 a.out
-----------------------------------------------------------------------
```

6kb for a simple hello world? that's a lot!

Even if I add other size optimization flags such as
-Wl,--gc-sections -fno-unwind-tables
-fno-asynchronous-unwind-tables -Os it just won't go below 6kb.

We will now progressively strip this program down by first getting
rid of the standard library and then learning how to invoke
syscalls without having to include any headers.

So how do we get rid of the standard library? If we try to compile
our current code with -nostdlib we will run into linker errors:

```
-----------------------------------------------------------------------
$> gcc -s -O2 -nostdlib hello.c
/usr/lib/gcc/x86_64-pc-linux-gnu/4.9.3/../../../../x86_64-pc-linux-
gnu/bin/ld: warning: cannot find entry symbol _start; defaulting to
0000000000400120
/tmp/ccTn8ClC.o: In function `main':
hello.c:(.text.startup+0xa): undefined reference to `puts'
collect2: error: ld returned 1 exit status
-----------------------------------------------------------------------
```

The linker is complaining about _start missing, which is what we
would expect from our previous debugging.

We also have a linker error on puts, which is to be expected since
it's a libc function. But how do we print "hello" without puts?

The linux kernel exposes a bunch of syscalls, which are functions
that user-space programs can enter to interact with the OS.
You can see a list of syscalls by running "man syscalls" or
visiting this site:
http://man7.org/linux/man-pages/man2/syscalls.2.html

How do we find out which syscall puts uses? We can either look
through the syscall list, or simply install strace to trace
syscalls and write a simple program that uses puts.

The strace method is extremely useful. If you don't know how to
do something with syscalls, do it with libc and then strace it to
see which syscalls it uses on the target architecture.

```
-----------------------------------------------------------------------
$> cat > puts.c << "EOF"
#include <stdio.h>

int main(int argc, char* argv[]) {
    puts("hello");

    return 0;
}
EOF

$> gcc puts.c
$> strace ./a.out > /dev/null
- stuff we don't care about -
write(1, "hello\n", 6)          = 6
exit_group(0)                   = ?
+++ exited with 0 +++
-----------------------------------------------------------------------
```

So it's using the write syscall.

Note how I pipe stdout to /dev/null in strace? That's because
strace output is in stderr and we don't want to have it mixed with
a.out's output.

Let's check the manpage for write:

```
$> man 2 write
SYNOPSIS
       #include <unistd.h>

       ssize_t write(int fd, const void *buf, size_t count);

DESCRIPTION
   write() writes up to count bytes from the buffer pointed
          buf to the file referred to by the file descriptor fd.
```

In linux, there are 3 standard file descriptors:
- stdin: used to pipe data into the program or to read user input.
- stdout: output
- stderr: alternate output for error messages

If we read "man stdout", we will see that they are simply defined as 0, 1 and 2.

So all we have to do is replace our puts with a write to stream 1 (stdout).

```
#include <unistd.h>

int main(int argc, char* argv[]) {
    write(1, "hello\n", 6);

    return 0;
}
```

Let's try to compile it again:

```
$> gcc -s -O2 -nostdlib hello.c
hello.c: In function ?main?:
hello.c:6:5: warning: ignoring return value of ?write?, declared
with attribute warn_unused_result [-Wunused-result]
     write(1, "hello\n", 6);
     ^
/usr/lib/gcc/x86_64-pc-linux-gnu/4.9.3/../../../../x86_64-pc-linux-
gnu/bin/ld: warning: cannot find entry symbol _start; defaulting to
0000000000400120
/tmp/ccJXwSsr.o: In function `main':
hello.c:(.text.startup+0x14): undefined reference to `write'
collect2: error: ld returned 1 exit status
```

Oh no! The "write" function is part of the standard library!
How do we invoke syscalls without having to link the standard lib?

Let's take a look at section "A.2.1 Calling Conventions" of the AMD64 ABI specification. If you are on i386 (32-bit), just follow along, we will port this to i386 soon in a moment.

If you're completely clueless about asm, you should still be able to understand once you see the example. I'm not that good at asm myself.

https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf

```
1. User-level applications use as integer registers for passing the
sequence %rdi, %rsi, %rdx, %rcx, %r8 and %r9. The kernel interface
uses %rdi, %rsi, %rdx, %r10, %r8 and %r9.

2. A system-call is done via the syscall instruction. The kernel
destroys registers %rcx and %r11.

3. The number of the syscall has to be passed in register %rax.

4. System-calls are limited to six arguments, no argument is passed
directly on the stack.

5. Returning from the syscall, register %rax contains the result of
the system-call. A value in the range between -4095 and -1
indicates an error, it is -errno.

6. Only values of class INTEGER or class MEMORY are passed to the
kernel.
```

In poor words, all we need to do is write an asm wrapper that:

- takes the syscall number followed by either pointers or integers

```
  as parameters
- sets rax to the syscall number
- sets rdi, rsi, rdx, r10, r8 and r9 to the parameters. calls that
  take less than 6 parameters will ignore the excess ones.
- executes "syscall"
- returns the contents of rax

Now if we read section 3.4 of the specification or the quick
cheatsheet at http://wiki.osdev.org/Calling_Conventions , we will
see that on AMD64 the registers used to pass parameters to regular
functions are almost the same as the syscalls, except for r10 which
is replaced with rcx. The return register is also the same (rax).

This means that our syscall wrapper will only be able to accept and
forward a maximum of 5 parameters (because the first parameter is
already being used to pass the syscall number).

We could use the stack to take more than 6 parameters, but let's
not make our lives more complicated when we don't even need to call
syscalls with 6 parameters yet.

The abi also states that:
---------------------------------------------------------------------
Registers %rbp, %rbx and %r12 through %r15 ?belong? to the calling
function and the called function is required to preserve their
values. In other words, a called function must preserve these
registers? values for its caller. Remaining registers ?belong? to
the called function. If a calling function wants to preserve such a
register value across a function call, it must save the value in
its local stack frame.
---------------------------------------------------------------------

Which means that we don't have to worry about saving and restoring
the values of rdi, rsi, rdx, r10, r8 and r9 inside of our syscall
wrapper, because it's up to the caller to save them and gcc will
take care of that (since we will be calling it from C code).

Putting it all together, this will be our syscall wrapper (in intel
syntax):
---------------------------------------------------------------------
mov rax,rdi    /* rax (syscall number)  = func param 1 (rdi) */
mov rdi,rsi    /* rdi (syscall param 1) = func param 2 (rsi) */
mov rsi,rdx    /* rsi (syscall param 2) = func param 3 (rdx) */
mov rdx,rcx    /* rdx (syscall param 3) = func param 4 (rcx) */
mov r10,r8     /* r10 (syscall param 4) = func param 5 (r8)  */
mov r8,r9      /* r8  (syscall param 5) = func param 6 (r9)  */
syscall        /* enter the syscall (return value will be in rax */
ret            /* return value is already in rax, we can return */
---------------------------------------------------------------------

How do we embed arbitrary asm into our program though? One way is
gcc inline assembler, but I personally find the syntax ugly.

We're going to write a .S file in GAS (GNU Assembler) syntax and
let gcc compile and link it with your hello.c .
---------------------------------------------------------------------
cat > hello.S << "EOF"
/* enable intel asm syntax without the % prefix for registers */
.intel_syntax noprefix

/* this marks the .text section of a PE executable, which contains
   program code */
.text
    /* exports syscall5 to other compilation units (files) */
    .globl syscall5

    syscall5:
        mov rax,rdi
        mov rdi,rsi
        mov rsi,rdx
        mov rdx,rcx
        mov r10,r8
        mov r8,r9
        syscall
        ret
EOF
---------------------------------------------------------------------

You can find syscalls numbers here:
http://betteros.org/ref/syscall.php
https://filippo.io/linux-syscall-table/

Or by simply letting the C preprocessor print it for you:
---------------------------------------------------------------------
$> printf "#include <sys/syscall.h>\nblah SYS_write" | \
  gcc -E - | grep blah
```

```
blah 1
-----------------------------------------------------------------
-E runs the preprocessor on the file, expanding all macros and
therefore replacing #define consts with their value, while - means
that we use stdin as input (which we pipe in from printf).
Then we just mark a line with blah so we can grep it, followed by
the constant we want to know.

Syscall numbers are usually named SYS_ followed by the syscall name
You can also add the -m32 flags to check values for 32-bit (i386).

Remember the prototype for write from earlier?
-----------------------------------------------------------------
ssize_t write(int fd, const void *buf, size_t count);
-----------------------------------------------------------------

ssize_t and size_t are types defined by unistd. A quick inspection
reveals that they are 64-bit integers and that the extra s in
ssize means signed:
-----------------------------------------------------------------
$> printf "#include <unistd.h>" | gcc -E - | grep size_t
typedef long int __blksize_t;
typedef long int __ssize_t;
typedef __ssize_t ssize_t;
typedef long unsigned int size_t;
-----------------------------------------------------------------

If we try -m32 we will also see that this will be a 32-bit integer
on 32-bit, which means that it's the same size as the
architecture's pointers. I like to call this kind of integer
intptr.

Now we can import syscall5 in hello.c and make a write function
that calls it:
-----------------------------------------------------------------
void* syscall5(
    void* number,
    void* arg1,
    void* arg2,
    void* arg3,
    void* arg4,
    void* arg5
);

typedef unsigned long int uintptr; /* size_t */
typedef long int intptr; /* ssize_t */

static intptr write(int fd, void const* data, uintptr nbytes) {
    return (intptr)
        syscall5(
            (void*)1, /* SYS_write */
            (void*)(intptr)fd,
            (void*)data,
            (void*)nbytes,
            0, /* ignored */
            0  /* ignored */
        );
}

int main(int argc, char* argv[]) {
    write(1, "hello\n", 6);

    return 0;
}
-----------------------------------------------------------------

See that (void*)(intptr) double cast on fd? If fd is 32-bit and
void* is 64-bit, we would get a warning that we are implicitly
casting it to a different size, so we need to explicitly specify
that we want that conversion by adding the intptr cast.

This should be done every time you cast to and from pointers when
the destination type is not guaranteed to be the same size as
pointers. Especially when targeting multiple architectures.

Also note how we cast the const qualifier away from data to avoid
a warning.

If we compile now, we are finally only missing _start!
-----------------------------------------------------------------
$> gcc -s -O2 -nostdlib hello.S hello.c
/usr/lib/gcc/x86_64-pc-linux-gnu/4.9.3/../../../../
x86_64-pc-linux-gnu/bin/ld: warning: cannot find entry symbol
_start; defaulting to 0000000000400120
-----------------------------------------------------------------
```

So, how do we define _start? Where do we get argc and argv from?
We need to know the initial state of registers and the stack.

Back to the AMD64 ABI document. In figure 3.9, we can see the
initial state of the stack:

```
0 to rsp: undefined
rsp          : argc         <- top of the stack (last pushed value)
rsp+8        : argv[0]
rsp+16       : argv[1]
rsp+24       : argv[2]
...          : ...
rsp+8*argc   : argv[argc - 1]
rsp+8+8*argc : 0
* more stuff we don't care about *
```

And right below it we have the initial state of the registers:

```
---------------------------------------------------------------------
%rbp: The content of this register is unspecified at process
      initialization time, but the user code should mark the
      deepest stack frame by setting the frame pointer to zero.

%rsp: The stack pointer holds the address of the byte with lowest
      address which is part of the stack. It is guaranteed to be
      16-byte aligned at process entry.

%rdx: a function pointer that the application should register with
      atexit (BA_OS).
---------------------------------------------------------------------
```

So we know that rbp must be zeroed and that rsp points to the top
of the stack. We don't care about rdx.

If you don't understand how the stack works, it's basically a
chunk of memory where data is appended (pushed) or retrieved (pop)
at the end.

In AMD64's convention we're actually prepending and removing data
at the beginning of the block of memory since the stack is said to
"grow downwards", which means that when we push something on the
stack, the stack pointer gets lower.

Since the ABI states that the stack pointer is 16-byte aligned, we
must remember always push data whose size is a multiple of 16. For
example, 2 64-bit integers are 16 bytes. It's often necessary to
either push useless data or simply align the stack pointer when
the pushed values don't happen to be aligned.

Putting it all together, our _start function needs to:
- zero rbp
- put argc into rdi (1st parameter for main)
- put the stack address of argv[0] into rsi (2nd param for main),
  which will be interpreted as an array of char pointers.
- align stack to 16-bytes
- call main

Here's our new hello.S:

```
---------------------------------------------------------------------
.intel_syntax noprefix
.text
    .globl _start, syscall5

    _start:
        xor rbp,rbp /* xoring a value with itself = 0 */
        pop rdi /* rdi = argc */
        /* the pop instruction already added 8 to rsp */
        mov rsi,rsp /* rest of the stack as an array of char ptr */

        /* zero the las 4 bits of rsp, aligning it to 16 bytes
           same as "and rsp,0xfffffffffffffff0" because negative
           numbers are represented as
           max_unsigned_value + 1 - abs(negative_num) */
        and rsp,-16
        call main
        ret

    syscall5:
        mov rax,rdi
        mov rdi,rsi
        mov rsi,rdx
        mov rdx,rcx
        mov r10,r8
        mov r8,r9
        syscall
        ret
---------------------------------------------------------------------
```

It finally compiles! It runs correctly, but we get a segmentation
fault when we exit:

```
$> gcc -s -O2 -nostdlib hello.S hello.c
$> ./a.out
hello
Segmentation fault
```

But why?

When we execute a call instruction, the return address (address of
the intruction to jump to after the function returns) is pushed
onto the stack implicitly and the ret instruction implicitly pops
it and jumps to it.

The _start function is very special, as it has no return address,
so our ret instruction in _start is trying to jump back to an
invalid memory location, executing garbage data as code or
triggering access violations.

We need to tell the OS to kill our process and never reach the ret
in _start. The syscall _EXIT(2) is just what we need:

```
$> man 2 _EXIT
NAME
       _exit, _Exit - terminate the calling process

SYNOPSIS
       #include <unistd.h>

       void _exit(int status);

       #include <stdlib.h>

       void _Exit(int status);

$> printf "#include <sys/syscall.h>\nblah SYS_exit" | \
  gcc -E - | grep blah
blah 60
```

The status code will simply be the return value of main, which is
stored in rax as we know.

New hello.S:

```
.intel_syntax noprefix
.text
    .globl _start, syscall5

    _start:
        xor rbp,rbp
        pop rdi
        mov rsi,rsp
        and rsp,-16
        call main

        mov rdi,rax /* syscall param 1 = rax (ret value of main) */
        mov rax,60 /* SYS_exit */
        syscall

        ret /* should never be reached, but if the OS somehow fails
               to kill us, it will cause a segmentation fault */

    syscall5:
        mov rax,rdi
        mov rdi,rsi
        mov rsi,rdx
        mov rdx,rcx
        mov r10,r8
        mov r8,r9
        syscall
        ret
```

Our program finally runs and terminates correctly! Let's give
ourselves a good pat on the back.

```
$> gcc -s -O2 -nostdlib hello.S hello.c
$> ./a.out
hello
```

Let's check the executable size now:

```
$> wc -c a.out
1008 a.out
```

We're almost below 1kb and it's 6 times smaller than before, but we
can shrink it further.

First of all, gcc generates unwind tables by default, which are
used for exception handling and other stuff we don't care about.

Let's turn those off:

```
$> gcc -s -O2 \
    -nostdlib \
    -fno-unwind-tables \
    -fno-asynchronous-unwind-tables \
    hello.S hello.c

$> wc -c a.out
736 a.out
```

Woah, we shaved almost 300 bytes off!

As a last step, we can check the executable for useless sections:

```
$> objdump -x a.out

a.out:        file format elf64-x86-64
a.out
architecture: i386:x86-64, flags 0x00000102:
EXEC_P, D_PAGED
start address 0x000000000040011a

Program Header:
    LOAD off   0x0000000000000000 vaddr 0x0000000000400000
         paddr  0x0000000000400000 align 2**21
         filesz 0x0000000000000153 memsz 0x0000000000000153
         flags r-x
   STACK off   0x0000000000000000 vaddr 0x0000000000000000
         paddr  0x0000000000000000 align 2**4
         filesz 0x0000000000000000 memsz 0x0000000000000000
         flags rwx
PAX_FLAGS off  0x0000000000000000 vaddr 0x0000000000000000
         paddr  0x0000000000000000 align 2**3
         filesz 0x0000000000000000 memsz 0x0000000000000000
         flags --- 2800

Sections:
Idx Name          Size      VMA               LMA               ...
  0 .text         0000005c  00000000004000f0  00000000004000f0   ...
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       00000007  000000000040014c  000000000040014c   ...
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .comment      0000002a  0000000000000000  0000000000000000   ...
                  CONTENTS, READONLY
SYMBOL TABLE:
no symbols
```

.text is the code
.rodata is Read Only data (such as the string "hello" in our case)

So we need both of these.

But what's that .comment section?

```
$> objdump -s -j .comment a.out

a.out:     file format elf64-x86-64

Contents of section .comment:
 0000 4743433a 20284765 6e746f6f 20342e39  GCC: (Gentoo 4.9
 0010 2e332070 312e352c 20706965 2d302e36  .3 p1.5, pie-0.6
 0020 2e342920 342e392e 3300               .4) 4.9.3.
```

Just information about the compiler, it seems. That's 1 byte for
every character of that string, let's get rid of it!

```
$> strip -R .comment a.out
$> wc -c a.out
624 a.out
```

There we go, we have achieved a nearly ten-fold size improvement
on our little hello world.

Let's set up a build script with all those compiler flags and let's
also make it output the executable with a proper name.

Also, I'm going to add the following useful flags:

-Wl,--gc-sections: get rid of any unused code sections

-fdata-sections: separate each function into its own code section.
                 this lets gc-sections do its job. these two
                 options combined will get rid of any dead code you
                 might accidentally leave in your program. it also
                 gets rid of unused functions in statically linked
                 libraries.

-fno-stack-protector: doesn't generate extra code to guard against
                      overflows overwriting the return address.

-Wa,--noexecstack: mark the stack memory as non-executable. this is
                   just extra security since we don't need to be
                   executing code off the stack's memory.

-fno-builtin: disable all builtin gcc functions (such as math
              routines and other stuff). we will implement them
              ourselves as needed.

-std=c89 -pedantic: follow the old c89 standard strictly. this
                    should force us to write code more compatible
                    with old compilers.

-Wall: enable all warnings.
-Werror: treat all warnings as error. can't let our code build with
         unchecked warnings.

```
$> cat > build.sh << "EOF"
#!/bin/sh

exename="hello"

gcc -std=c89 -pedantic -s -O2 -Wall -Werror \
    -nostdlib \
    -fno-unwind-tables \
    -fno-asynchronous-unwind-tables \
    -fdata-sections \
    -Wl,--gc-sections \
    -Wa,--noexecstack \
    -fno-builtin \
    -fno-stack-protector \
    hello.S hello.c \
    -o $exename \
\
&& strip -R .comment $exename
EOF

$> chmod +x ./build.sh
$> ./build.sh
$> wc -c hello
624 hello
$> ./hello
hello
```

As you might have noticed, we are doing a lot of useless mov's in
that syscall5 wrapper on syscalls that take less than 5 parameters.

Let's make one wrapper for each parameter count. This will increase
performance slightly at the cost of a slightly bigger executable.

You are free to remove the ones you don't use once you finish
prototyping your program.

New hello.S

```
.intel_syntax noprefix
.text
    .globl _start, syscall,
    .globl syscall1, syscall2, syscall3, syscall4, syscall5

    _start:
        xor rbp,rbp
        pop rdi
        mov rsi,rsp
        and rsp,-16
```

```
        call main
        mov rdi,rax
        mov rax,60 /* SYS_exit */
        syscall
        ret

    syscall:
        mov rax,rdi
        syscall
        ret

    syscall1:
        mov rax,rdi
        mov rdi,rsi
        syscall
        ret

    syscall2:
        mov rax,rdi
        mov rdi,rsi
        mov rsi,rdx
        syscall
        ret

    syscall3:
        mov rax,rdi
        mov rdi,rsi
        mov rsi,rdx
        mov rdx,rcx
        syscall
        ret

    syscall4:
        mov rax,rdi
        mov rdi,rsi
        mov rsi,rdx
        mov rdx,rcx
        mov r10,r8
        syscall
        ret

    syscall5:
        mov rax,rdi
        mov rdi,rsi
        mov rsi,rdx
        mov rdx,rcx
        mov r10,r8
        mov r8,r9
        syscall
        ret
```

Now we can change our write function to use syscall3 instead.

We will also change argv in our main to be char const* since we probably won't be modifying it. This is normally not allowed on the standard C library, but we aren't using it :^).

Using the syscall numbers directly is a bit hard to read so let's also make a header with all the syscall numbers we use:

```
$> cat > syscalls.h << "EOF"
#define SYS_write 1
#define SYS_exit  60
EOF
```

We will also define the syscall number as uintptr so that we don't need to cast to void*.

new hello.c

```
#include "syscalls.h"

typedef unsigned long int uintptr;
typedef long int intptr;

void* syscall3(
    uintptr number,
    void* arg1,
    void* arg2,
    void* arg3
);

static intptr write(int fd, void const* data, uintptr nbytes) {
    return (uintptr)
```

```
        syscall3(
            SYS_write,
            (void*)(intptr)fd,
            (void*)data,
            (void*)nbytes
        );
}

int main(int argc, char const* argv[]) {
    write(1, "hello\n", 6);

    return 0;
}
```

We can include headers in .S files, so let's also include it in hello.S

```
#include "syscalls.h"

.intel_syntax noprefix
.text
    .globl _start, syscall,
    .globl syscall1, syscall2, syscall3, syscall4, syscall5

    _start:
        xor rbp,rbp
        pop rdi
        mov rsi,rsp
        and rsp,-16
        call main
        mov rdi,rax
        mov rax,SYS_exit
        syscall
        ret

...
```

Having to pass the string length every time is annoying, so let's implement our own strlen and puts.

I'm also going to make a "internal" alias for static, which makes it easier to search for static functions, rather than static variables, in a large codebase. I got this idea from Casey Muratori from handmade hero.

```
#include "syscalls.h"

typedef unsigned long int uintptr;
typedef long int intptr;

#define internal static

void* syscall3(
    uintptr number,
    void* arg1,
    void* arg2,
    void* arg3
);

/* ------------------------------------------------------------ */

#define stdout 1

internal intptr write(int fd, void const* data, uintptr nbytes) {
    return (uintptr)
        syscall3(
            SYS_write,
            (void*)(intptr)fd,
            (void*)data,
            (void*)nbytes
        );
}

/* ------------------------------------------------------------ */

internal uintptr strlen(char const* str) {
    char const* p;
    for (p = str; *p; ++p);
    return p - str;
}

internal uintptr puts(char const* str) {
    return write(stdout, str, strlen(str));
```

```
}

/* ---------------------------------------------------------- */

int main(int argc, char const* argv[]) {
    puts("hello\n");

    return 0;
}
```

If you don't understand my strlen function, it's pretty simple: C strings are null-terminated (the byte after the last character is zero), so I just iterate the characters through a pointer until I find a zero byte, and then I subtract the current position from the beginning of the string.

libc does all kinds of crazy tricks to optimize this for large strings, which I haven't looked into.

As you can see, I've also separated the code into sections with those spacer comments for readability. I grouped all the syscall wrappers together, followed by utility functions, followed by the program's code.

Now we have a nice framework for AMD64 programs, but we're not going to stop here. We're going to set this up to also cross compile for i386, which is a very common architecture in low-end servers (such as the one I host my gopher mirror on).

```
########################################################################
                          Porting to i386
########################################################################
Let's move all the AMD64-specific code into a dedicated folder.
------------------------------------------------------------------------
$> mkdir amd64
$> mv hello.S amd64/start.S
$> mv syscalls.h amd64/
------------------------------------------------------------------------
```

Now we can make a architecture-specific main.c where we define the integer types and main, which just calls hello_run, or whatever you want to name your program's entry point. This file includes hello.c just before main.

I also make it define AMD64 in case we need to do platform checking in the code. Platform specific code should be kept separated whenever possible, though.

```
------------------------------------------------------------------------
$> cat > amd64/main.c << "EOF"
#define AMD64
#include "syscalls.h"

typedef unsigned long int  u64;
typedef unsigned int       u32;
typedef unsigned short int u16;
typedef unsigned char      u8;

typedef long int    i64;
typedef int         i32;
typedef short int   i16;
typedef signed char i8;

typedef i64 intptr;
typedef u64 uintptr;

#include "../hello.c"

int main(int argc, char const* argv[]) {
    return hello_run(argc, argv);
}
EOF
------------------------------------------------------------------------
```

Yes, you can include .c files, which just get pasted into the file. This results in a single compilation unit even though we have multiple files, which speeds up compilation (unless your project is massive) and saves us the pain of typing every filename in our build script. This is yet another tick I got from Casey.

By the way, you can check integer types on any architecture with the usual gcc preprocessor trick:

```
------------------------------------------------------------------------
$> printf "#include <stdint.h>" | gcc -E - | grep int64
typedef long int int64_t;
```

```
typedef unsigned long int uint64_t;

$> printf "#include <stdint.h>" | gcc -E - | grep int32
typedef int int32_t;
typedef unsigned int uint32_t;

$> printf "#include <stdint.h>" | gcc -E - | grep int16
typedef short int int16_t;
typedef unsigned short int uint16_t;

$> printf "#include <stdint.h>" | gcc -E - | grep int8
typedef signed char int8_t;
typedef unsigned char uint8_t;
----------------------------------------------------------------------

And for the size of pointers, you can write a simple program that
printfs sizeof(void*).

hello.c will now look like this (remember, we moved the integer
definitions to main.c and renamed main to hello_run, and
syscalls.h is already included in main.c):

----------------------------------------------------------------------
#define internal static

void* syscall3(
    uintptr number,
    void* arg1,
    void* arg2,
    void* arg3
);

/* ------------------------------------------------------------ */

#define stdout 1

internal intptr write(int fd, void const* data, uintptr nbytes) {
    return (uintptr)
        syscall3(
            SYS_write,
            (void*)(intptr)fd,
            (void*)data,
            (void*)nbytes
        );
}

/* ------------------------------------------------------------ */

internal uintptr strlen(char const* str) {
    char const* p;
    for (p = str; *p; ++p);
    return p - str;
}

internal uintptr puts(char const* str) {
    return write(stdout, str, strlen(str));
}

/* ------------------------------------------------------------ */

internal int hello_run(int argc, char const* argv[]) {
    puts("hello\n");

    return 0;
}
----------------------------------------------------------------------

Modify the build script to follow the new structure:
----------------------------------------------------------------------
#!/bin/sh

exename="hello"

gcc -std=c89 -pedantic -s -O2 -Wall -Werror \
    -nostdlib \
    -fno-unwind-tables \
    -fno-asynchronous-unwind-tables \
    -fdata-sections \
    -Wl,--gc-sections \
    -Wa,--noexecstack \
    -fno-builtin \
    -fno-stack-protector \
    amd64/start.S amd64/main.c \
    -o $exename \
\
&& strip -R .comment $exename
```

```
-----------------------------------------------------------------------

Now we can create the main.c for i386:
-----------------------------------------------------------------------
$> mkdir i386
$> cat > i386/main.c << "EOF"
#define I386
#include "syscalls.h"

typedef unsigned long long int u64;
typedef unsigned int           u32;
typedef unsigned short int     u16;
typedef unsigned char          u8;

typedef long long int i64;
typedef int           i32;
typedef short int     i16;
typedef signed char   i8;

typedef i32 intptr;
typedef u32 uintptr;

#include "../hello.c"

int main(int argc, char const* argv[]) {
    return hello_run(argc, argv);
}
EOF
-----------------------------------------------------------------------

Note how intptr is defined as a 32-bit integer and u64 is long
long on 32-bits.

Let's now grab syscall numbers for i386 and throw them into
syscalls.h:
-----------------------------------------------------------------------
$> printf "#include <sys/syscall.h>\nblah SYS_write" \
  | gcc -m32 -E - | grep blah
blah 4

$> printf "#include <sys/syscall.h>\nblah SYS_exit" \
  | gcc -m32 -E - | grep blah
blah 1

$> cat > i386/syscalls.h << "EOF"
#define SYS_write 4
#define SYS_exit  1
EOF
-----------------------------------------------------------------------

We need to write a i386 start.S and you guessed it, it's time to
look at the ABI specification once again!

http://www.sco.com/developers/devspecs/abi386-4.pdf

This time I will just summarize the differences from amd64:

- Registers are 32-bit so we push 4 bytes at a time.

- The stack is aligned to 4 bytes, but we will still align it to
  16 bytes because it can improve performance by preventing
  misaligned SSE accesses (according to glibc).

- ebp needs to be zeroed (32-bit version of rbp)

- esp is the stack pointer (32-bit version of rsp)

- Return values for functions and syscalls are in eax

- The instruction to enter syscalls is "int 0x80"

- Syscall parameters are passed in ebx, ecx, edx, esi, edi, ebp

- Function parameters are passed entirely through the stack by
  pushing them in reverse order, which means that we will be able
  to access them sequentially every 4 bytes on the stack.
  VERY IMPORTANT DIFFERENCE. We won't be using registers to pass
  parameters to main anymore nor to pull parameters in syscall
  wrappers.

- Functions are expected to preserve ebx, esi, edi, ebp, esp on
  their own VERY IMPORTANT! we will have to save and restore these
  registers manually in our syscall wrappers!

- Function callers are expected to clean up the parameters off the
  stack after the call. VERY IMPORTANT
```

- As explained earlier, the return address is implicitly pushed on
  the stack so the function parameters will start at esp+4.

In short, our _start will look something like this:

```
xor ebp,ebp

pop esi /* argc */
mov ecx,esp /* argv */

/* 16-byte stack alignment is not mandatory here but
   according to glibc it improves SSE performance */
and esp,-16

/* push garbage to align to 16 bytes */
push 0xb16b00b5
push 0xb16b00b5
push ecx /* argv */
push esi /* argc */
call main
add esp,16
/* on i386 it's up to the caller to clean up the stack. we can
   either pop them into scratch registers or just add the total
   size of the parameters in bytes to the stack pointer */

mov ebx,eax
mov eax,SYS_exit
int 0x80
ret
```

... and our syscall5 wrapper will look like this:

```
push ebx
push esi
push edi
mov eax,[esp+4+12]
mov ebx,[esp+8+12]
mov ecx,[esp+12+12]
mov edx,[esp+16+12]
mov esi,[esp+20+12]
mov edi,[esp+24+12]
int 0x80
pop edi
pop esi
pop ebx
ret
```

See how I'm pushing registers on the stack to preserve them to then pop them (in reverse order since it's LIFO)? That's very important on i386.

Also, you might be wondering what's going on with the esp offsets. You have to keep in mind that every time I push a register on the stack, esp is decremented by 4, so I have to skip the registers I pushed on the stack (3 registers = 12 bytes) to get to the parameters. Don't forget that the return address is also on the stack, so parameters start at + 4.

And here's our complete i386 start.S

```
$> cat > i386/start.S << "EOF"
#include "syscalls.h"

.intel_syntax noprefix
.text
    .globl _start, syscall
    .globl syscall1, syscall2, syscall3, syscall4, syscall5

    _start:
        xor ebp,ebp
        pop esi
        mov ecx,esp
        and esp,-16
        push 0xb1gb00b5
        push 0xb1gb00b5
        push ecx
        push esi
        call main
        add esp,16
        mov ebx,eax
        mov eax,SYS_exit
        int 0x80
        ret
```

```
    syscall:
        mov eax,[esp+4]
        int 0x80
        ret

    syscall1:
        push ebx
        mov eax,[esp+4+4]
        mov ebx,[esp+8+4]
        int 0x80
        pop ebx
        ret

    syscall2:
        push ebx
        mov eax,[esp+4+4]
        mov ebx,[esp+8+4]
        mov ecx,[esp+12+4]
        int 0x80
        pop ebx
        ret

    syscall3:
        push ebx
        mov eax,[esp+4+4]
        mov ebx,[esp+8+4]
        mov ecx,[esp+12+4]
        mov edx,[esp+16+4]
        int 0x80
        pop ebx
        ret

    syscall4:
        push ebx
        push esi
        mov eax,[esp+4+8]
        mov ebx,[esp+8+8]
        mov ecx,[esp+12+8]
        mov edx,[esp+16+8]
        mov esi,[esp+20+8]
        int 0x80
        pop esi
        pop ebx
        ret

    syscall5:
        push ebx
        push esi
        push edi
        mov eax,[esp+4+12]
        mov ebx,[esp+8+12]
        mov ecx,[esp+12+12]
        mov edx,[esp+16+12]
        mov esi,[esp+20+12]
        mov edi,[esp+24+12]
        int 0x80
        pop edi
        pop esi
        pop ebx
        ret
EOF
-----------------------------------------------------------------------
```

Now we need to modify our build script to handle multiple architectures.

I will just make the script take the arch subfolder name as a parameter.

This is not enough though, because each architecture will have some extra compiler flags. For example, on i386 we need -m32 to ensure a 32-bit build even on amd64 dev machines, as well as -Wno-long-long which suppresses a warning about 64 bit integers being a nonstandard gcc extension on 32-bit.

We will make our build script source a flags.sh script in the architecture-specific folder which just exports COMPILER_FLAGS with all the extra stuff it wants.

```
-----------------------------------------------------------------------
$> cat > build.sh << "EOF"
#!/bin/sh

exename="hello"
archname=${1:-amd64} # if not specified, default to amd64
```

```
# if flags.sh exists in the arch folder, source it
if [ -e $archname/flags.sh ]; then
    source $archname/flags.sh
fi

gcc -std=c89 -pedantic -s -O2 -Wall -Werror \
    -nostdlib \
    -fno-unwind-tables \
    -fno-asynchronous-unwind-tables \
    -fdata-sections \
    -Wl,--gc-sections \
    -Wa,--noexecstack \
    -fno-builtin \
    -fno-stack-protector \
    $COMPILER_FLAGS \
    $archname/start.S $archname/main.c \
    -o $exename \
\
&& strip -R .comment $exename
EOF

$> cat > i386/flags.sh << "EOF"
#!/bin/sh

export COMPILER_FLAGS="-m32 -Wno-long-long"
EOF
----------------------------------------------------------------------

Now we can compile both architectures easily with minimal code
redundancy:
----------------------------------------------------------------------
$> wc -c hello
720 hello
$> ./hello
hello
$> ./build.sh i386
$> wc -c hello
608 hello
$> ./hello
hello
----------------------------------------------------------------------

And there you have it! You now have a nice framework to develop
libc-free programs.

As you can see, the 32-bit executable is slightly smaller. This is
mostly because pointers are half as large compared to 64-bit.

######################################################################
                        Legacy syscalls on i386
######################################################################
There are a few things you should be extremely careful with when
dealing with syscalls, especially when targeting multiple
architectures.

Some syscalls, such as stat, might return their stuff in a struct.
Be extremely careful to check the struct layout and size of the
types used, because it will often change drastically between
architectures.

----------------------------------------------------------------------
$> man 2 stat
NAME
       stat, fstat, lstat, fstatat - get file status

SYNOPSIS
       #include <sys/types.h>
       #include <sys/stat.h>
       #include <unistd.h>

       int stat(const char *pathname, struct stat *buf);
       int fstat(int fd, struct stat *buf);
       int lstat(const char *pathname, struct stat *buf);

$> printf "#include <sys/stat.h>" | gcc -E - | grep -A 1 "int stat"
extern int stat (const char *__restrict __file,
   struct stat *__restrict __buf) __attribute__ ((__nothrow__ ,
   __leaf__)) __attribute__ ((__nonnull__ (1, 2)));
----------------------------------------------------------------------

----------------------------------------------------------------------
$> printf "#include <sys/stat.h>" \
  | gcc -E - | grep -A 60 "struct stat"
struct stat {
    __dev_t st_dev;
```

```
    __ino_t st_ino;
    __nlink_t st_nlink;
    __mode_t st_mode;
    __uid_t st_uid;
    __gid_t st_gid;
    int __pad0;
    __dev_t st_rdev;
    __off_t st_size;
    __blksize_t st_blksize;
    __blkcnt_t st_blocks;
# 91 "/usr/include/bits/stat.h" 3 4
    struct timespec st_atim;
    struct timespec st_mtim;
    struct timespec st_ctim;
# 106 "/usr/include/bits/stat.h" 3 4
    __syscall_slong_t __glibc_reserved[3];
# 115 "/usr/include/bits/stat.h" 3 4
};

$> printf "#include <sys/stat.h>" | gcc -E - \
  | grep '__dev_t\|__ino_t\|__nlink_t\|__mode_t\|__uid_t\|__gid_t'
typedef unsigned long int __dev_t;
typedef unsigned int __uid_t;
typedef unsigned int __gid_t;
typedef unsigned long int __ino_t;
typedef unsigned int __mode_t;
typedef unsigned long int __nlink_t;

$> printf "#include <sys/stat.h>" | gcc -E - \
  | grep '__blksize_t\|__blkcnt_t\|__syscall_slong_t\|__off_t'
typedef long int __off_t;
typedef long int __blksize_t;
typedef long int __blkcnt_t;
typedef long int __syscall_slong_t;

$> printf "#include <sys/stat.h>" | gcc -E - \
  | grep -A 10 "struct timespec"
struct timespec {
    __time_t tv_sec;
    __syscall_slong_t tv_nsec;
};

$> printf "#include <sys/stat.h>" | gcc -E - | grep "__time_t"
typedef long int __time_t;
---------------------------------------------------------------------

---------------------------------------------------------------------
$> printf "#include <sys/stat.h>" \
  | gcc -m32 -E - | grep -A 60 "struct stat"
struct stat {
    __dev_t st_dev;
    unsigned short int __pad1;
    __ino_t st_ino;
    __mode_t st_mode;
    __nlink_t st_nlink;
    __uid_t st_uid;
    __gid_t st_gid;
    __dev_t st_rdev;
    unsigned short int __pad2;
    __off_t st_size;
    __blksize_t st_blksize;
    __blkcnt_t st_blocks;
# 91 "/usr/include/bits/stat.h" 3 4
    struct timespec st_atim;
    struct timespec st_mtim;
    struct timespec st_ctim;
# 109 "/usr/include/bits/stat.h" 3 4
    unsigned long int __glibc_reserved4;
    unsigned long int __glibc_reserved5;
};

$> printf "#include <sys/stat.h>" | gcc -m32 -E - \
  | grep '__dev_t\|__ino_t\|__nlink_t\|__mode_t\|__uid_t\|__gid_t'
__extension__ typedef __u_quad_t __dev_t;
__extension__ typedef unsigned int __uid_t;
__extension__ typedef unsigned int __gid_t;
__extension__ typedef unsigned long int __ino_t;
__extension__ typedef unsigned int __mode_t;
__extension__ typedef unsigned int __nlink_t;

$> printf "#include <sys/stat.h>" \
  | gcc -m32 -E - | grep '__u_quad_t'
__extension__ typedef unsigned long long int __u_quad_t;

$> printf "#include <sys/stat.h>" | gcc -m32 -E - \
  | grep '__blksize_t\|__blkcnt_t\|__syscall_slong_t'
```

```
__extension__ typedef long int __off_t;
__extension__ typedef long int __blksize_t;
__extension__ typedef long int __blkcnt_t;
__extension__ typedef long int __syscall_slong_t;

$> printf "#include <sys/stat.h>" | gcc -m32 -E - \
  | grep -A 10 "struct timespec"
struct timespec {
    __time_t tv_sec;
    __syscall_slong_t tv_nsec;
};

$> printf "#include <sys/stat.h>" | gcc -m32 -E - | grep "__time_t"
__extension__ typedef long int __time_t;
```

As you can see, the stat struct is substantially different for
i386 and amd64 and the contained types are also different in size.

This is not all there is to it though. Some syscalls have multiple
versions of them with different structs for historical reasons, and
gcc might wrap them in some weird way, using its own struct.

stat is one of them. Suppose you use the above structs and assume
libc, stat struct is right.
Let's make a simple program that stats a file and dumps the stat
struct to stdout for us to inspect.

These are the files:

```
$> cat amd64/syscalls.h
#define SYS_write 1
#define SYS_stat  4
#define SYS_exit  60

$> cat i386/syscalls.h
#define SYS_write 4
#define SYS_stat 106
#define SYS_exit  1

$> cat stat.c
#define internal static

void* syscall2(
    uintptr number,
    void* arg1,
    void* arg2
);

void* syscall3(
    uintptr number,
    void* arg1,
    void* arg2,
    void* arg3
);

/* ------------------------------------------------------------ */

#define stdout 1

internal intptr write(int fd, void const* data, uintptr nbytes) {
    return (uintptr)
        syscall3(
            SYS_write,
            (void*)(intptr)fd,
            (void*)data,
            (void*)nbytes
        );
}

typedef u64 dev_t;
typedef intptr syscall_slong_t;
typedef intptr time_t;

typedef struct {
    time_t sec;
    syscall_slong_t nsec;
}
timespec;

typedef struct {
    dev_t dev;
#ifdef I386
    u16 __pad1;
#endif
    uintptr ino;
```

```
    uintptr nlink;
    u32 mode;
    u32 uid;
    u32 gid;
#ifdef AMD64
    int __pad0;
#endif
    dev_t rdev;
#ifdef I386
    u16 __pad2;
#endif
    intptr size;
    intptr blksize;
    intptr blocks;
    timespec atim;
    timespec mtim;
    timespec ctim;
#ifdef AMD64
    syscall_slong_t __glibc_reserved[3];
#else
    u32 __glibc_reserved4;
    u32 __glibc_reserved5;
#endif
}
stat_info;

internal int stat(char const* path, stat_info* s) {
    return (int)(intptr)
        syscall2(
            SYS_stat,
            (void*)path,
            s
        );
}

/* ---------------------------------------------------------- */

internal int stat_run(int argc, char const* argv[]) {
    stat_info si;

    if (stat("/etc/hosts", &si) == 0) {
        write(stdout, &si, sizeof(stat_info));
    }

    return 0;
}
```

Now if we hexdump output from amd64 and i386, we will see that
something is not quite right on i386:

```
$> ./build.sh
$> ./stat | hexdump -C
00000000  12 08 00 00 00 00 00 00  50 59 0a 00 00 00 00 00
00000010  01 00 00 00 00 00 00 00  a4 81 00 00 00 00 00 00
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00000030  bc 04 00 00 00 00 00 00  00 10 00 00 00 00 00 00
00000040  08 00 00 00 00 00 00 00  24 b2 e9 57 00 00 00 00
00000050  d1 f4 e1 2f 00 00 00 00  e8 d8 5e 57 00 00 00 00
00000060  a0 3a b4 24 00 00 00 00  e8 d8 5e 57 00 00 00 00
00000070  20 c8 0f 25 00 00 00 00  00 00 00 00 00 00 00 00
00000080  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00000090
$> ./build.sh i386
$> ./stat | hexdump -C
00000000  12 08 00 00 50 59 0a 00  a4 81 01 00 00 00 00 00
00000010  00 00 00 00 bc 04 00 00  00 10 00 00 08 00 00 00
00000020  24 b2 e9 57 d1 f4 e1 2f  e8 d8 5e 57 a0 3a b4 24
00000030  e8 d8 5e 57 20 c8 0f 25  00 00 00 00 00 00 00 00
00000040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00000050  00 00 00 00 00 00 00 00
00000058
```

We know dev_t is a 64-bit integer from our previous investigations,
so why is other stuff being packed after the 4th byte? The first
8 bytes of the structs should be the same as amd64!

If you scroll through the stat manpage, you will find this:

```
Over  time, increases in the size of the stat structure have led to
three successive versions of  stat():  sys_stat()  (slot  __NR_old?
stat), sys_newstat() (slot __NR_stat), and sys_stat64() (slot
__NR_stat64) on 32-bit platforms such as i386. The first two ver?
sions  were  already  present  in  Linux 1.0 (albeit with different
names); the last was added in Linux 2.4.  Similar remarks apply for
```

```
fstat() and lstat().

The  kernel-internal  versions  of the stat structure dealt with by
the different versions are, respectively:

      __old_kernel_stat
             The original structure, with  rather  narrow  fields,
             and no padding.

      stat   Larger  st_ino  field    and  padding  added to various
             parts of the structure to allow for future expansion.

      stat64 Even larger st_ino field, larger  st_uid  and    st_gid
             fields to accommodate the Linux-2.4 expansion of UIDs
             and GIDs to  32  bits,  and  various  other  enlarged
             fields  and further padding in the structure.    (Vari?
             ous padding bytes were eventually consumed  in  Linux
             2.6,  with  the  advent  of  32-bit  device  IDs  and
             nanosecond components for the timestamp fields.)

The glibc stat() wrapper function hides these details from applica?
tions, invoking the most recent version of the system call provided
by the kernel, and repacking the returned information  if  required
for old binaries.
-----------------------------------------------------------------------

So it's likely that glibc is tampering with stat instead of just
forwarding the syscall.

You can actually check this by writing a small libc stat test and
using strace to trace syscalls:
-----------------------------------------------------------------------
$> cat > stattest.c << "EOF"
#include <sys/stat.h>

int main() {
    struct stat s;
    stat("/etc/hosts", &s);
    return 0;
}
EOF

$> gcc -m32 stattest.c
$> strace ./a.out
execve("./a.out", ["./a.out"], [/* 83 vars */]) = 0
[ Process PID=22487 runs in 32 bit mode. ]
... stuff we don't care about ...
stat64("/etc/hosts", {st_mode=S_IFREG|0644, st_size=1212, ...}) = 0
exit_group(0)                           = ?
+++ exited with 0 +++
-----------------------------------------------------------------------

Yep, as expected, the stat call is getting translated to stat64!

So how do we fix this? By not trusting libc headers and digging
into the kernel headers (which I found by googling the kernel
struct names):
-----------------------------------------------------------------------
$> printf "#include <asm/stat.h>" \
  | gcc -m32 -E - | grep -A 30 "struct stat"
struct stat {
    unsigned long st_dev;
    unsigned long st_ino;
    unsigned short st_mode;
    unsigned short st_nlink;
    unsigned short st_uid;
    unsigned short st_gid;
    unsigned long st_rdev;
    unsigned long st_size;
    unsigned long st_blksize;
    unsigned long st_blocks;
    unsigned long st_atime;
    unsigned long st_atime_nsec;
    unsigned long st_mtime;
    unsigned long st_mtime_nsec;
    unsigned long st_ctime;
    unsigned long st_ctime_nsec;
    unsigned long __unused4;
    unsigned long __unused5;
};
-----------------------------------------------------------------------

That's a very different than what glibc headers were telling us!
There is no padding and st_dev is 4 bytes instead of 8, as well
as a lot of other fields having smaller sizes.
```

What about the 64-bit version of it?

```
$> printf "#include <asm/stat.h>" \
  | gcc -E - | grep -A 30 "struct stat"
struct stat {
    __kernel_ulong_t st_dev;
    __kernel_ulong_t st_ino;
    __kernel_ulong_t st_nlink;

    unsigned int st_mode;
    unsigned int st_uid;
    unsigned int st_gid;
    unsigned int __pad0;
    __kernel_ulong_t st_rdev;
    __kernel_long_t st_size;
    __kernel_long_t st_blksize;
    __kernel_long_t st_blocks;

    __kernel_ulong_t st_atime;
    __kernel_ulong_t st_atime_nsec;
    __kernel_ulong_t st_mtime;
    __kernel_ulong_t st_mtime_nsec;
    __kernel_ulong_t st_ctime;
    __kernel_ulong_t st_ctime_nsec;
    __kernel_long_t __unused[3];
};
```

This one seems to have the correct layout, except that some of the values are unsigned rather than signed.

Here's our fixed stat struct:

```
typedef uintptr dev_t;
typedef intptr syscall_slong_t;
typedef uintptr syscall_ulong_t;
typedef uintptr time_t;

typedef struct {
    time_t sec;
    syscall_ulong_t nsec;
}
timespec;

typedef struct {
    dev_t dev;
    uintptr ino;
#ifdef AMD64
    uintptr nlink;
    u32 mode;
    u32 uid;
    u32 gid;
    u32 __pad0;
#else
    u16 mode;
    u16 nlink;
    u16 uid;
    u16 gid;
#endif
    dev_t rdev;
    uintptr size;
    uintptr blksize;
    uintptr blocks;
    timespec atim;
    timespec mtim;
    timespec ctim;
#ifdef AMD64
    syscall_slong_t __unused[3];
#else
    u32 __unused4;
    u32 __unused5;
#endif
}
stat_info;
```

Now we can run it again and verify that the struct is properly populated in both architectures (I added comments to show where fields are, those aren't actually part of hexdump)

```
$> ./stat | hexdump -C
00000000  12 08 00 00 00 00 00 00  50 59 0a 00 00 00 00 00
         |         dev           |          ino           |

00000010  01 00 00 00 00 00 00 00  a4 81 00 00 00 00 00 00
         |        nlink          |    mode    |    uid    |
```

```
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
         |    gid    |  __pad0   |          rdev         |

00000030  bc 04 00 00 00 00 00 00  00 10 00 00 00 00 00 00
         |          size         |        blksize        |

00000040  08 00 00 00 00 00 00 00  24 b2 e9 57 00 00 00 00
         |         blocks        |       atim.sec        |

00000050  d1 f4 e1 2f 00 00 00 00  e8 d8 5e 57 00 00 00 00
         |       atim.nsec       |       mtim.sec        |

00000060  a0 3a b4 24 00 00 00 00  e8 d8 5e 57 00 00 00 00
         |       mtim.nsec       |       ctim.sec        |

00000070  20 c8 0f 25 00 00 00 00  00 00 00 00 00 00 00 00
         |       ctim.nsec       |      __unused[0]      |

00000080  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
         |      __unused[1]      |      __unused[2]      |

00000090

$> ./build.sh i386
$> ./stat | hexdump -C
00000000  12 08 00 00 50 59 0a 00  a4 81 01 00 00 00 00 00
          |    dev   |    ino    | mode |nlink| uid | gid |

00000010  00 00 00 00 bc 04 00 00  00 10 00 00 08 00 00 00
          |   rdev   |   size    |   blksize  |   blocks  |

00000020  24 b2 e9 57 d1 f4 e1 2f  e8 d8 5e 57 a0 3a b4 24
          | atim.sec | atim.nsec |  mtim.sec  | mtim.nsec |

00000030  e8 d8 5e 57 20 c8 0f 25  00 00 00 00 00 00 00 00
          | ctim.sec | ctim.nsec | __unused4  | __unused5 |

00000040
```

In short, try getting structs from kernel headers instead of libc.

```
########################################################################
                      Legacy sockets on i386
########################################################################
```

Another thing you should be aware of, is that some syscalls might work entirely differently on i386 because of historical reasons.

Socket syscalls are a perfect example. i386 doesn't have SYS_accept and as far as I know the other socket syscalls are also not guaranteed to exist.

Instead, i386 multiplexes all socket syscalls through a single syscall named "socketcall", which takes an additional param which specifies which socket operation we want do to, (accept, connect, etc...) followed by the usual syscall params that we find on amd64.

Also, parameters for socketcall are passed through a void* array, so the socketcall syscall just takes two parameters: the call number and the pointer to the parameters array.

Googling the socketcall numbers was a bit difficult, but I eventually found them in linux/net.h.

```
$> printf "#include <sys/syscall.h>\nblah SYS_accept" \
  | gcc -m32 -E - | grep blah
blah SYS_accept

$> man socketcall
SYNOPSIS
int socketcall(int call, unsigned long *args);

DESCRIPTION
socketcall()  is  a common kernel entry point for the socket system
calls. call determines which socket function to invoke. args
points to a block containing the actual arguments, which are passed
through to the appropriate call.

User programs should call the appropriate functions by their  usual
names. Only standard library implementors and kernel hackers need
to know about socketcall().

$> printf "#include <linux/net.h>\nblah SYS_SOCKET" \
```

```
  | gcc -m32 -E - | grep blah
blah 1

$> printf "#include <linux/net.h>\nblah SYS_CONNECT" \
  | gcc -m32 -E - | grep blah
blah 3
```

Here's an example socket application for i386 and amd64 that connects to sdf.org's gopherspace (192.94.73.15:70) and dumps the output for the root folder.

I got the sockaddr_in struct from netinet/in.h and the socket constants from sys/socket.h

---

```
$> cat amd64/syscalls.h
#define SYS_read    0
#define SYS_write   1
#define SYS_close   3
#define SYS_socket  41
#define SYS_connect 42
#define SYS_exit    60

$> cat i386/syscalls.h
#define SYS_read       3
#define SYS_write      4
#define SYS_close      6
#define SYS_exit       1
#define SYS_socketcall 102

$> cat socket.c
#define internal static

void* syscall1(
    uintptr number,
    void* arg1
);

void* syscall2(
    uintptr number,
    void* arg1,
    void* arg2
);

void* syscall3(
    uintptr number,
    void* arg1,
    void* arg2,
    void* arg3
);

/* ------------------------------------------------------------ */

#define stdout 1
#define stderr 2

internal void close(int fd) {
    syscall1(SYS_close, (void*)(intptr)fd);
}

internal intptr write(int fd, void const* data, uintptr nbytes) {
    return (uintptr)
        syscall3(
            SYS_write,
            (void*)(intptr)fd,
            (void*)data,
            (void*)nbytes
        );
}

internal intptr read(int fd, void* data, intptr nbytes) {
    return (intptr)
        syscall3(
            SYS_read,
            (void*)(intptr)fd,
            data,
            (void*)nbytes
        );
}

#define AF_INET 2
#define SOCK_STREAM 1
#define IPPROTO_TCP 6

typedef struct {
```

```
    u16 family;
    u16 port; /* NOTE: this is big endian!!!!!!! use flip16u */
    u32 addr; /* this is also big endian */
    u8  zero[8];
}
sockaddr_in;

#ifdef SYS_socketcall
/* i386 multiplexes socket calls through socketcall */
#define SYS_SOCKET      1
#define SYS_CONNECT     3

internal int socketcall(u32 call, void* args) {
    return (int)(intptr)
        syscall2(
            SYS_socketcall,
            (void*)(intptr)call,
            args
        );
}
#endif

internal int socket(u16 family, i32 type, i32 protocol) {
#ifndef SYS_socketcall
    return (int)(intptr)
        syscall3(
            SYS_socket,
            (void*)(intptr)family,
            (void*)(intptr)type,
            (void*)(intptr)protocol
        );
#else
    void* args[3];
    args[0] = (void*)(intptr)family;
    args[1] = (void*)(intptr)type;
    args[2] = (void*)(intptr)protocol;

    return socketcall(SYS_SOCKET, args);
#endif
}

internal int connect(int sockfd, sockaddr_in const* addr) {
#ifndef SYS_socketcall
    return (int)(intptr)
        syscall3(
            SYS_connect,
            (void*)(intptr)sockfd,
            (void*)addr,
            (void*)sizeof(sockaddr_in)
        );
#else
    void* args[3];
    args[0] = (void*)(intptr)sockfd;
    args[1] = (void*)addr;
    args[2] = (void*)sizeof(sockaddr_in);

    return socketcall(SYS_CONNECT, args);
#endif
}

/* ------------------------------------------------------------ */

internal intptr strlen(char const* str) {
    char const* p;
    for(p = str; *p; ++p);
    return p - str;
}

internal intptr fputs(int fd, char const* str) {
    return write(fd, str, strlen(str));
}

/* reverses byte order of a 16-bit integer (0x1234 -> 0x3412) */
internal u16 flip16u(u16 v) {
    return (v << 8) | (v >> 8);
}

/* ------------------------------------------------------------ */

#define BUFSIZE 512

internal int socket_run(int argc, char const* argv[]) {
    int res = 0; /* return code */

    int fd;
    u8 ip_raw[] = { 192, 94, 73, 15 }; /* ip in big endian order */
```

```
    u32* pip = (u32*)ip_raw; /* pointer to ip as a 32-bit int */
    sockaddr_in a;

    intptr n;
    u8 buf[BUFSIZE];

    /* set up sockaddr struct with desired ip & port */
    a.family = AF_INET;
    a.port = flip16u(70);
    a.addr = *pip;

    for (n = 0; n < 8; ++n) {
        a.zero[n] = 0;
    }

    /* create a new socket */
    fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (fd < 0) {
        fputs(stderr, "socket failed\n");
        return 1;
    }

    /* connect to sdf.org */
    if (connect(fd, &a) < 0) {
        fputs(stderr, "connect failed\n");
        res = 1;
        goto cleanup;
    }

    /* request folder / */
    fputs(fd, "/\r\n");

    /* read chunks of BUFSIZE bytes and relay them to stdout until
       there is nothing left to read or the socket errors out */
    while (1) {
        n = read(fd, buf, BUFSIZE);
        if (n <= 0) {
            break;
        }

        if (write(stdout, buf, n) != n) {
            fputs(stderr, "write failed\n");
            res = 1;
            break;
        }
    }

    if (n < 0) {
        fputs(stderr, "read failed\n");
        res = 1;
    }

cleanup:
    /* make sure to not leave a dangling socket file descriptor */
    close(fd);

    return res;
}
-----------------------------------------------------------------------

And as you can see, we are running flawlessly on both architectures:w

-----------------------------------------------------------------------
$> ./build.sh && ./socket
iWelcome to the SDF Public Access UNIX System .. est. 1987...

$> ./build.sh i386 && ./socket
iWelcome to the SDF Public Access UNIX System .. est. 1987...
-----------------------------------------------------------------------

#########################################################################
                            Conclusion
#########################################################################
I hope this guide got you interested in understanding what happens
at the lowest level and knowing your programming language and OS
beyond the standard library! Have fun! I will add more tricks if
I come up with new ones.


---
```