CSE 410: Systems Programming

Effective Systems Programming

Ethan Blanton

Department of Computer Science and Engineering
University at Buffalo

Effective Systems Programming

Effective systems programming is about knowing your system.

We've spent a lot of time talking about POSIX.

- POSIX covers a lot of system
- Not all systems are POSIX, however[1]

Most systems will use many the same paradigms, if not details.

Take the time to learn the system.

[1] sadly

Corners of C

C has deep, dark corners that we have not explored.

Some of them are valuable, many of them are dangerous.

Many undefined and implementation-dependent behaviors are in those corners.

And that's not to even mention C11!

Debugging

As you've seen this semester debugging systems is difficult.

We have looked at:

- Compiler warnings/errors
- Printing to stderr
- `gdb`
- `valgrind`
- optionally compiling code
- …

There are more tools available!

Learn them.

Some Common System Variations

- Word sizes
- Signedness of `char`
- Availability of floating point operations
- Path separator for filenames
- Availability of `fork()`
- Unavailability of threads or processes
- Availability of `mmap()`

There are many, but you'll see these often, particularly on:

- Embedded systems
- Non-POSIX systems with a POSIX compatibility layer

Word Sizes

If your code has to be portable, use sized integers everywhere!

- `int32_t`
- `uintptr_t`
- `ptrdiff_t`

Even if it isn't required for correctness, it communicates intention.

Signedness of `char`

The signedness of `char` comes up surprisingly often.

It's often related to casting `int` to `char` or vice-versa.

Examples:

- Range checks for raw bytes
- Loops with comparison to zero

To be safe, use `int8_t` or `uint8_t` for binary data.

Floating Point

All C compilers[2] will compile floating point.

However:

- It may not be IEEE 754
- It may be software emulated

The former you might not notice (x86-64 isn't!), the latter you will.

Most systems programs should avoid floating point entirely.

Use integer or fixed point math when possible.

---

[2] Real C compilers that I'm aware of, that is

Path Separators

All POSIX systems use forward slash as their path separator.

Some other systems use other characters.[3]

C doesn't define this and doesn't deal with it.

Portable programs have to do a lot of work for this.

Consider using a compatibility library like GLib if this is a concern.

[3] Inexplicably

Fork, threads, and processes

Embedded systems, in particular, may not have `fork()`.

Some non-POSIX systems do not have `fork()`.

Many such systems:

- Will have POSIX threads
- May have `posix_spawn()`

If you don't have any of these, C has a (painful) answer:

- `setjmp()`
- `longjmp()`

Memory Mappings

Most non-POSIX systems will not have `mmap()`.

This includes many embedded systems with a POSIX layer.

Sometimes this means there is no virtual memory.

Sometimes it means you need to use a different interface.

Shared memory may be available through some other interface.

Variadic Functions

We only briefly mentioned variadic functions.

This is a way to write a function with a variable number of arguments.

The declaration syntax is simple:

```
void func(type firstarg, ...);
```

Every variadic function must accept at least one named argument.

There are restrictions on the type of the last named argument.

The number of arguments must be determined at run time.

Variable Argument Lists

```
#include <stdarg.h>

typedef /* system-dependent */ va_list;
void va_start(va_list ap, parameter);
type va_arg(va_list ap, type);
void va_end(va_list ap);
```

A function must:

- Call va_start() first
- Call va_arg() zero or more times
- Call va_end() before returning

Example Variadic Function

```c
/* Print all string arguments to fp, end on NULL */
void example(FILE *fp, ...) {
    va_list ap;
    char *arg;

    va_start(ap, fp);
    while ((arg = va_arg(ap, char *)) != NULL) {
        fprintf(fp, "%s\n", arg);
    }
    va_end(ap);
}
```

Preprocessor Macro Arguments

Preprocessor macro arguments can be manipulated with #.

A single # turns an argument into a string:

```
#define logptr(x) printf("%s: %p\n", #x, x)

int var;
logptr(&var);
```

Output:

```
&var: 0x7fff06ee7e6c
```

Preprocessor Macro Arguments

Preprocessor macro arguments can be manipulated with #.

Two # concatenate C tokens:

```
#define printvar(x) printf("var%d: %d\n", x, var ## x)

int var1 = 42;
int var2 = 31337;
printvar(1);
printvar(2);
```

Output:

```
var1: 42
var2: 31337
```

Variadic Macros

In addition to variadic functions, C99 has variadic macros.

They are dangerous but powerful.

- Dangerous because they make code even harder to understand than regular macros
- Powerful because they enable calling variadic functions, iterations on lists, *etc.*

```
#define DEBUG(format, ...) \
    fprintf(stderr, "%s:%d" format, __FILE__, \
            __LINE__, ##__VA_ARGS__)
```

Bit-field Integers

Adjacent integers in structs can be bit fields.

Bit fields have explicitly specified width in bits.

```
struct Bitfields {
    unsigned int onenibble:4;
    unsigned int byteandahalf:12;
};
```

This struct might be as small as two bytes.

The precise behavior and layout of bitfields is implementation-defined.

Named and Ordered Initialization

This isn't "deep C", but it's very useful for readability!

- Arrays can used numbered initializers
- Structures can use named initializers

```
/* All entries except 2, 3, and 7 are 0 */
int array[100] = { [2] = 3, [3] = 5, [7] = 13 };

/* Any fields except question and answer are 0 */
struct Something s = {
    .question = "Life, the Universe, and Everything",
    .answer = 42
}
```

Goto

A controversial feature in any language, C has goto.

When used judiciously, it can be very powerful.
(Most kernels are full of gotos!)

Its syntax is simple:

```
    i=10;
loop:
    i--;
    if (i > 0) goto loop;
```

That's a terrible goto, don't do it.

Judicious Goto

Goto is often used for cleanup:

```
    int fd = open("somefile", O_RDONLY);
    char *buf = malloc(BUFSIZE);

    if (do_something(fd, buf) < 0) goto cleanup;
    if (something_else(fd, buf) < 0) goto cleanup;
    /* ... */

cleanup:
    close(fd);
    free(buf);
```

System Logs

System logging functions can be valuable.

POSIX systems have syslog() for this purpose:

```
#include <syslog.h>

void openlog(const char *ident, int option, int
    facility);
void syslog(int priority, const char *format, ...);
void closelog(void);
```

This sends log messages to a logging daemon.

Generating Syslogs

For example, the following code:

```
openlog("example", 0, LOG_USER);
syslog(LOG_DEBUG, "the widget is frobnicated");
closelog();
```

This prints to /var/log/syslog:

```
Dec  4 21:13:28 westruun example: the widget is
    frobnicated
```

You would normally `openlog()` once, then `closlog()` before exiting.

Obvious Markers

It is common to use obvious values as markers.

These markers can be easily found by eye examining memory.

Examples:

- `0xfeedface`
- `0xdeadbeef`
- `0x01020304`
- `0x00badbad`
- `0xdeadc0de`

In addition, `0xaaaaaaaa` and `0x55555555` are alternating 1/0.

Forced Crashes

There are many ways to force a C program to dump core:

- `*NULL = 0;`
- `abort();`
- Send SIGABRT to a process with `kill`
- Press `C-\` at the terminal
- ...

This can be handy when an error condition is rare.

Assertions

A particular form of forced crash is an assertion.

```
#include <assert.h>

void assert(expression);
```

If `expression` evaluates to false, the program crashes.

Use assertions to test preconditions and postconditions.

Don't use assertions to check user input.

Turn off all but the most critical assertions unless debugging.

Use the Compiler

The compiler knows a lot about C.

Make it work for you, not against you:

- Compile with `-Wall -Werror` (and maybe `-Wextra`)
- Use structs and unions, not macros and pointer math
- Use functions, not macros
- Use enums, not `#define`s
- Make typedefs
- Silence warnings before digging too deep!

Your Editor

Find a good editor, and trust it.

If it thinks something is hinky, figure out why.

For example:

- It wants to indent funny
- It colors a variable name unexpectedly
- It can't find a completion
- …

This may mean things like:

- You've misplaced braces
- You're shadowing a system variable
- *etc.*

Congratualations

Congratulations, you’re systems programmers now.

I hope you’ve had a great semester.

Please fill out course evaluations.

See you Wednesday, December 12, at 08:00.

License