Everything you ever wanted to know about "hello, world"* (*but were afraid to ask.)

Brooks Davis
SRI International
<brooks.davis@sri.com>

1 Abstract

The first example in the classic book "The C Programming Language" by Kernighan and Ritchie[1] is in fact a remarkably complete test of the C programming language. This paper provides a guided tour of a slightly more complex program, where `printf()` is called with multiple arguments. Along the way from the initial processes's call to `exec()` to the final `_exit()`, we'll tour the program loading code in the kernel, the basics of system-call implementation, the implementation of the memory allocator, and of course `printf()`. We'll also touch on localization, a little on threading support, and a brief overview of the dynamic linker.

2 Introduction

The first example in the classic K&R C book is a simple program that prints the text "hello, world" on a single line and exits. It's seemingly simple and straightforward (once the programmer understands that `\n` means *newline*). In reality though, it compiles to over 550KiB on MIPS64! This paper explores the complexities behind the simple `helloworld` binary and uncovers the impressive engineering required to make this program work in a modern C environment.

```
#include <stdio.h>
main() {
  printf("hello, world\n");
}
```

The example above shows the classic K&R version of `helloworld`. Unfortunately, it won't even compile with a modern compiler in C11 mode. In practice we need to specify a return type (or at least `void` for returning nothing) and arguments. In theory the `main()` function is defined to return an integer and take two arguments for the command line argument list; however, compilers tend to accept this alternative form for compatibility sake.

```
#include <stdio.h>
void
main(void)
{
  printf("hello, world\n");
}
```

One of the more interesting features of the `printf()` function is that it takes a variable number of arguments. Varidac functions such as `printf()` are both interesting and error prone, so it's useful to understand how they work. As such, we'll use the following enhanced version of `helloworld` with a string argument and an integer argument. The resulting program prints "hello, world 123" on a single line.

```
int
main(void)
{
  const char hello[] =
    "hello, world";
  printf("%s %d\n", hello, 123);
  return (0);
}
```

2.1 A minimal version

Before we dive into the version above, let's consider the simplest program with the same result. For ex-

ample, this C program avoids all the complex formatting routines in `printf()` and writes a string to standard output directly via the `write()` system call:

```
void
main(void)
{
  const char *hello[] =
    "hello, world 123\n";
  write(1, hello, sizeof(hello));
  exit(0);
}
```

Taking things one step further, the following MIPS64 assembly does the same without the need for any standard startup code:

```
        .text
        .global __start
        .ent __start
__start:
        li $a0,  1
        dla $a1, hello
        li $a2, 13
        li $v0, 4
  # write(1, "hello, world 123\n",
  #   16)
        syscall
        li $a0, 0
        li $v0, 1
        syscall # exit(0)
        .end __start
.data hello:

.ascii "hello, world 123\n"
```

This version assembles down to 9 instructions and is less than one 1KiB in size. The vast majority of that size is due to ELF headers and MIPS platform support bits like specify details of the target architecture and required features.

3 The process life cycle

The life of a traditional Unix process is fairly straightforward. An existing process creates a copy of itself via the `fork()` system call. This copy is identical to the parent process, with exceptions in the parent/child relationship in the process tree and by extension the return value of `fork()`: `0` in the child and the process ID of the child in the parent. The child process has a copy of the memory of the parent process. (In practice, writable pages are marked copy-on-write to avoid unnecessary duplication.) Pages marked `shared` will be available in both processes. All of this is interesting and complicated, but beyond the scope of this paper, because the next thing the shell (or other process running `helloworld`) does is call `exec()` – which wipes away all this state to create a fresh process. Unless the process triggers an unhandled signal and is killed, it eventually calls `_exit()` and is terminated.

Modern UNIX systems may employ the `posix_spawn()` function to create a new process with a new executable, in one fell swoop eliminating the need for `exec()` to tear down the memory mappings `fork()` just created.

3.1 The exec system call

The `execve()` system call underpins all variants of `exec()` and has the prototype:

```
int execve(const char *path,
  char *const argv[],
  char *const envp[]);
```

Figure 1 on the following page displays a flame chart of the execution of the `sys_execve()` system call. The `execve()` system call's job is to open the file to be executed, clear the memory image of the current process, map the file to be executed into the process space, set up a stack with argument and environment vectors embedded, and finally return to user space with argument and program counter registers set up to call the `__start()` function.

`execve()` accomplishes this in a few phases. First, the `exec_copyin_args()` function copies the program name, argument vector, and environment vector into kernel memory. This varies between ABIs, as the pointers in the argument arrays may be of different sizes (for example, 32-bit pointers in an i386 binary running on an amd64 system). Once complete, `do_execve()`, the ABI-independent core of `sys_execve()` is called via the thin wrapper `kern_execve()`. It first uses `namei()` to resolve the path of the executable into a file, and then calls `exec_check_permissions()` – which verifies that the caller has permission to open the file and then opens it. The `exec_map_first_page()`

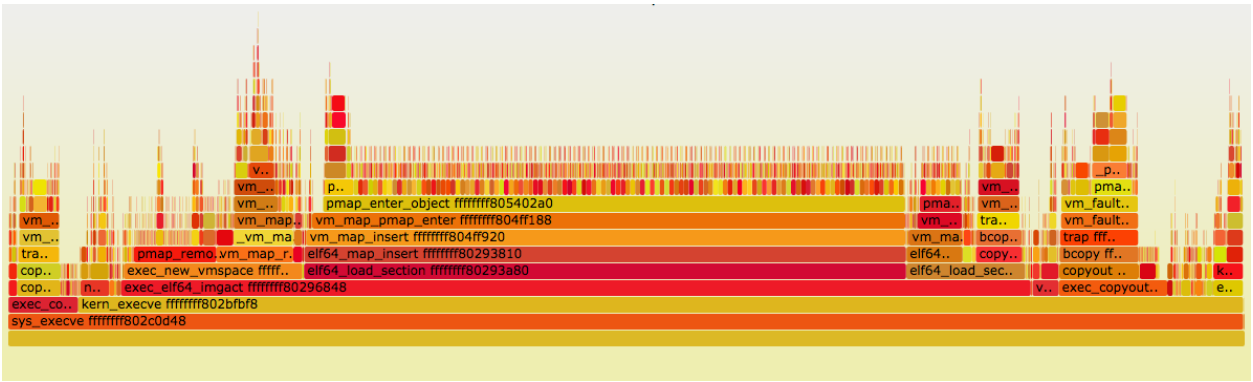

Figure 1: Flame chart of `sys_execve()` execution

function then does what you would expect: allowing the program headers to be parsed and the binary format determined (or in the case of a script, the interpreter to be found, but that is outside the scope of this paper). Next, the credentials of the new process are updated if the binary is marked setuid or setgid.

A binary-specific loading function referred to as an image activator is then called to parse and configure the process address space. One interesting side-effect of the use of `exec_map_first_page()` is that current image activators all require that any information required to set up the address space be stored in the first page of the binary. In practice, this isn't a significant limitation.

In the case of our MIPS n64 `helloworld`, the image activator is `exec_elf64_imgact()` – since we're mapping a native 64-bit ELF binary. (Confusingly, you won't find this function anywhere in the kernel, as it is declared via multiple macros in `kern/imgact_elf.c` as `__CONCAT(exec_, __elfN(imgact))(struct image_params *imgp)`.) `exec_elf64_imgact()` examines the ELF branding information to determine the ABI of the new process and finds the `sysvec` structure associated with the ABI. It then calls `exec_new_vmspace()`, which removes all the page mappings copied from the parent process by `fork()` creating an empty address space. It then maps the default stack and returns. `exec_elf64_imgact()` then maps segments of type `PT_LOAD` into the address space with appropriate permissions, the ELF auxiliary arguments vector is allocated, and portions derived directly from the ELF file are added.

At this point all the initial memory mappings for the process are set up. The kernel now needs to copy out the argument array, environment array, and ELF auxiliary arguments. This is performed by the `exec_copyout_strings()` function. The function `exec_copyout_strings()` is somewhat overloaded. In addition to copying out the program path, and the argument and environment strings, it reserves space for some auxiliary argument values and constructs the arrays of pointers to the aforementioned strings. It also copies out `sigcode` (the return trampoline for single handlers), generates and copies out the stack canary, and copies out a list of page sizes. Next `do_execve()` calls the `elf64_freebsd_fixup()` function, which updates the pointers in the previous allocated auxiliary argument array and writes `argc` (the number of program arguments) to the top of the stack. The layout of an n64 ABI process's memory at the time execution begins can be see in Figure 2 on the next page

Finally, `exec_setregs()` is called to initialize the state of the registers when `execve()` returns to the process. On MIPS, this means zeroing the register set, setting the stack pointer to the top of the default stack (less all the values copied out above), the program counter to the address of `__start()`, a few other MIPS specific registers to appropriate values, and the first argument register to the location of `argc`. (As a holdover from the port of MIPS support from NetBSD, we also set the fourth argument register to point to `ps_strings`, but that is unused in FreeBSD code.)

At this point `do_execve()` performs a bit of cleanup and then returns. We return to

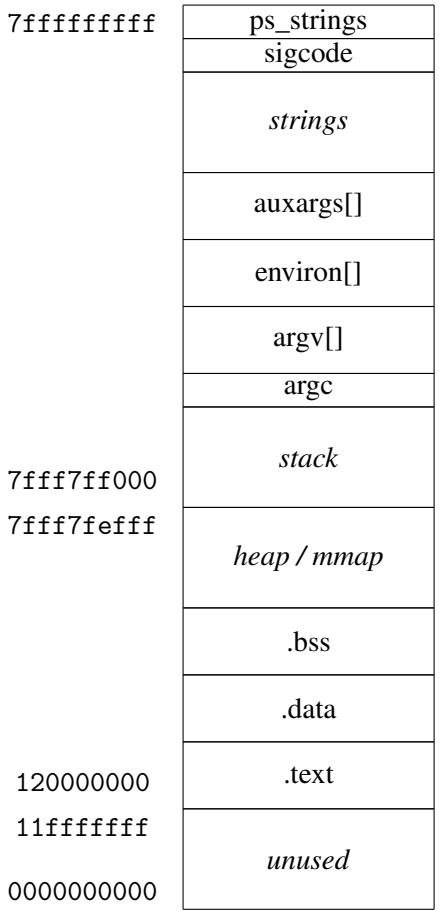

Figure 2: MIPS n64 static process memory

`kern_execve()` and `sys_execve()`, finally returning to userspace from the `execve()` system call.

3.2 Entering `__start`

As previously mentioned, and contrary to naïve expectations, statically linked programs begin in the `__start()` function, not in `main()`. This function handles initialization of process state that is not easily compiled into the static binary, and which may change over time – and thus is not suitable for embedding in the kernel. The `__start()` function and related infrastructure account for a considerable portion of the size of the `helloworld` binary. They are linked with the main program by the compiler, and are typically hidden from the programmer. For example, a typical command to link a `helloworld.o` object file into a `helloworld` executable looks like:

```
cc -static -o helloworld helloworld.o
```

Under the hood, the compiler actually calls the linker (`ld`) a s follows:

```
ld -EB -melf64btsmip_fbsd -Bstatic \
  -o helloworld /usr/lib/crt1.o \
  /usr/lib/crti.o /usr/lib/crtbeginT.o \
  -L/usr/lib helloworld.o \
  --start-group -lgcc -lgcc_eh -lc \
  --end-group \
  /usr/lib/crtend.o /usr/lib/crtn.o
```

These files include the implementation of `__start()` and an assortment of startup code responsible for invoking various initialization mechanisms. A brief explanation of the contents of the various files is shown in Table 1 on the following page.

In addition to invoking these initialization mechanisms, `__start()` calls `main()`; if main returns, `__start()` calls `exit()` with `main()`'s return value. The majority of time in `__start()` before entering `main()` is in `_init_tls()`, which sets up thread-local storage. This thread-local storage setup primarily consists of initializing data structures required for `malloc()` to allocate a buffer. In a program as simple as `helloworld`, this initialization takes more time than the actual `main()` function.

File	Purpose
`crt1.o`	Contains `__start()` function which initializes process environment and calls `main()`.
`crti.o`	Entry points for old style `_init()` and `_fini()` functions.

`crtbegin.o`
`crtbeginS.o`
`crtbeginT.o`
	Declares `.ctor` and `.dtor` constructor and destructor sections. Declares functions to call constructors and destructors.

`crtend.o`	NULL terminates `.ctor` and `.dtor` sections.
`crtn.o`	Trailers for `_init()` and `_fini()` functions.

Table 1: C Runtime files and their purposes

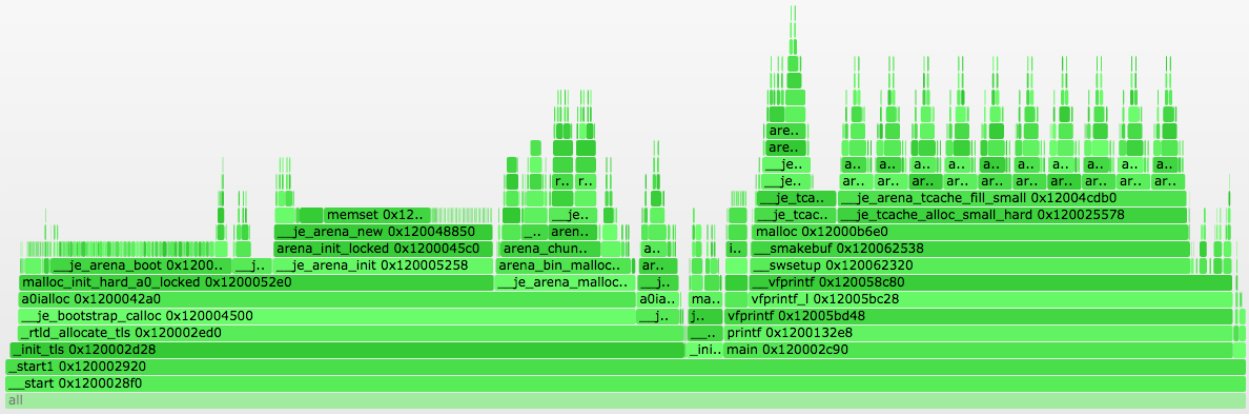

Figure 3: Flame chart of `__start()` execution

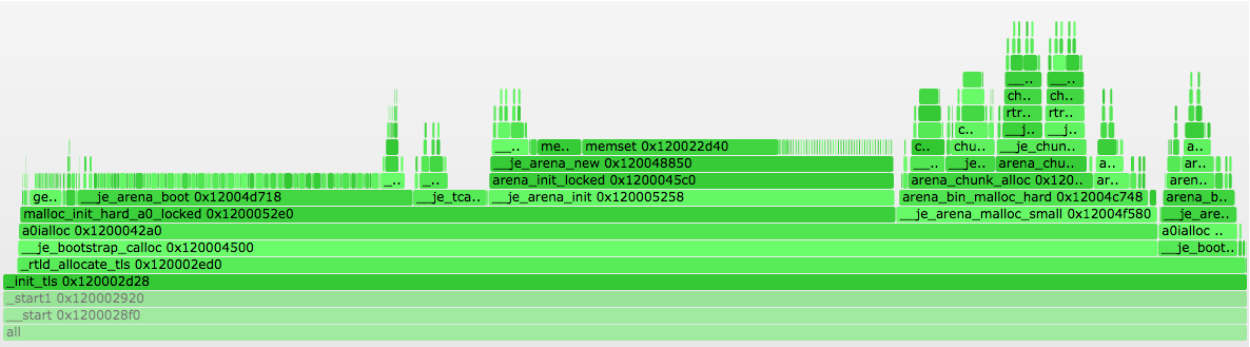

Figure 4: Flame chart of `_init_tls()` execution

The `_init_tls()` function's job is to allocate and initialize the thread-local storage area for the first thread. Examining the call graph in Figure 4 on the previous page, it is clear that most of the time is spent in `__je_bootstrap_calloc()`, which in turn spends most of its time in malloc initialization code. This is because the TLS area may need to be expanded in the future, due to `dlopen()` including new libraries that use TLS; when that happens, memory will be released with `free()`, and a new segment allocated. Thus, virtually every program linked against `libc` initializes the `malloc()` subsystem before `main()`. After memory is allocated, it needs to be initialized. The majority of thread-local variables are initialized to zero, so the use of `calloc()` takes care of their initialization; however, some have other default values defaults (e.g., a pointer to a default object, `-1` for an invalid file descriptor). For those, default values are stored in the executable at a location accessible via the ELF auxiliary arguments. The `_init_tls()` function thus needs to locate the auxiliary arguments vector. There is no standard way to do this, but the known layout of the stack provides a mechanism.

```
Elf_Addr *sp;
sp = (Elf_Addr *) environ;
while (*sp++ != 0)
  ;
aux = (Elf_Auxinfo *) sp;
```

Because the auxiliary argument vector lies just beyond the environment array, the code starts at the beginning of the argument array and proceeds to walk on past the end to find the auxiliary argument vector. This relies on behavior that is undefined in the C language.

Once the argument vector is located and cast to an appropraite type, `_init_tls()` walks it to find the ELF program headers; it then walks the header list to find the `PT_TLS` segment, copies values to the beginning of the previously allocated TLS entry, and sets the TLS pointer. On MIPS, the TLS pointer is set via `sysarch()` system calls, and historically retrieved by `sysarch()` as well. For setting, this is not important (as changes are infrequent). However, retrieving the pointer this way is another matter. Having to perform a system call every time a thread-local variable is accessed is quite expensive, since functions as common as `malloc()` and (as we see later) `printf()` use thread-local storage. On modern MIPS processors, the TLS point is stored in a special register that is updated on context switch or when set via the `sysarch()`.

Following TLS initialization, the `handle_static_init()` function calls various initialization functions. For historical reasons, there are more of these than strictly necessary. First, a `.pre_init_array` section contains a list of function pointers. Each is called in turn. Next, the `.init` section is a concatenation of all `.init` sections in the program and is the `_init` function. As previously mentioned, the prolog of `_init()` is defined in `crti.o` and the epilog in `crtn.o`. The body of the function is arbitrary machine code from various object files and libraries. This functionality is deprecated, and little use is made of it in modern software. However, the GNU startup code uses it to call code to handle constructors stored in the `.ctors` section and register a handler for the destructors in the `.dtors` section via the `atexit()` function. The `.ctors` section works similarly to the `.pre_init_array` section. After `_init()` returns, the `.init_array` section is handled identically to the `.pre_init_array` section.

All in all, there are more initialization methods than necessary, but we're stuck with them all on current platforms. New platform ports should consider removing `.init`/`.fini` sections entirely, and having `handle_static_init()` or its equivalent process `.ctor` and `.dtor` entries directly.

3.3 Calling `main()`

The `main()` function seen in Figure 5 on the following page simply calls `printf()` before returning to `__start()` in order to exit. `printf()` is implemented as a call to `vfprintf()`, which in turn retrieves the current thread's locale via `__get_locale()` and passes it to the function `vfprintf_l()`. `vfprintf_l()` finally calls `__vfprintf()` to do the actual work. One might reasonably ask, why do we need the locale in `printf()`? In the common case, the answer is that the format string might indicate thousands separators or decimal-points in numbers (commas and periods respectively in English). The locale is required to select the proper character. As shown in the flame

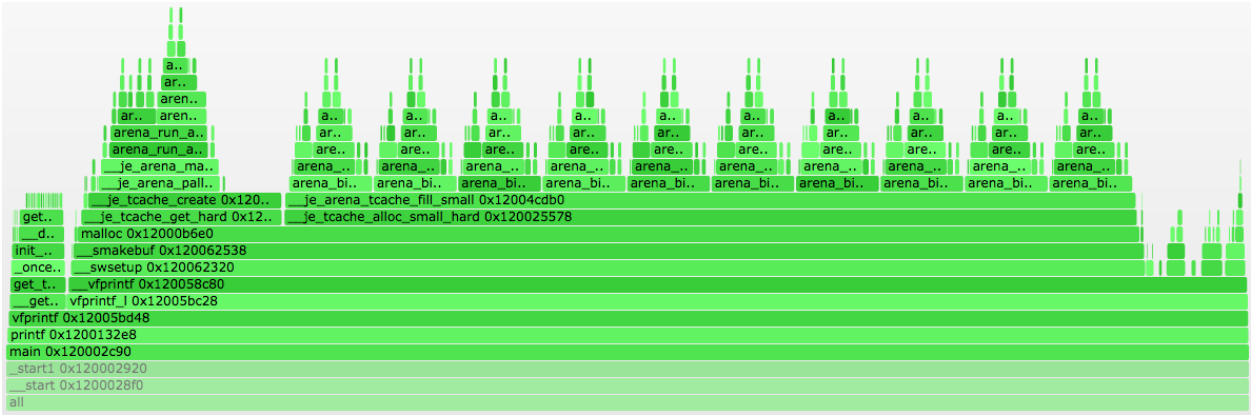

Figure 5: Flame chart of `main()` execution

chart, the majority of time in `printf()` is actually spent allocating an internal buffer in `__swsetup()`, which is beyond the scope of this paper.

The pertinent part of `__vfprintf()` seen in Figure 6 on the next page parses the format string in a loop, emitting output each time a format variable is parsed via the `__sprint()` function. Because output in this case goes to the buffered `STDOUT` file stream, actual output occurs only when the buffer is full or when a newline is detected. In this example, that occurs when outputting the last element of the format string.

The `__sprint()` function seen in Figure 7 on the following page scans the data to be output with `memchr()`, and locates a newline character. The data is output to the buffer as usual, except that at the end the buffer is flushed by the `__fflush()` function – which ultimately culminates in a call to the `write()` system call outputting:

```
hello, world 123
```

In our example implementation `main()` returns `0` to `__start()`, which calls `exit()`.

3.4 Exiting

Processes ultimately exit by being killed due to a signal or by calling the `_exit()` system call. Programs, including those that return to the standard `__start()` function, call the `exit()` library function to do this. The `exit()` function visible in Figure 8 on the next page is responsible for more than just calling the `_exit()` system call. In particular, it needs to call any finalizers or destructors. In the case of `helloworld`, the only one that is particularly interesting is `_cleanup()`. `_cleanup()` is called by any program that uses a buffered file stream to walk the list of buffered streams and flush any with outstanding data. Since our `STDOUT()` stream was flushed by the newline terminating the format string passed to `printf()`, `_cleanup()` just needs to observe that no data is buffered.

After `_cleanup()`, `exit()` calls `_exit()`, which terminates the process, and `helloworld` is complete.

4 Dynamic linking

Dynamically linked programs are similar to statically linked programs with a few key differences. First, statically linked programs directly include all code they may call. Second, statically linked programs are generally compiled to be invoked at a single predetermined userspace address (which is known as being non-relocatable). Third, for dynamically linked programs, the kernel loads both the the program (as is done for static programs) and the runtime linker. When `execve()` returns to userspace, it returns calling `rtld_start()` in the runtime linker rather than `__start()` in the program.

The runtime linker relocates itself, analyzes and relocates the program if necessary, and loads and relocates dynamically linked libraries. In the case of `helloworld`, this means loading and relocating `libc`, the C standard library. After relocation has occurred (at considerable cost in our trivial program), the linker calls `__start()` and execution

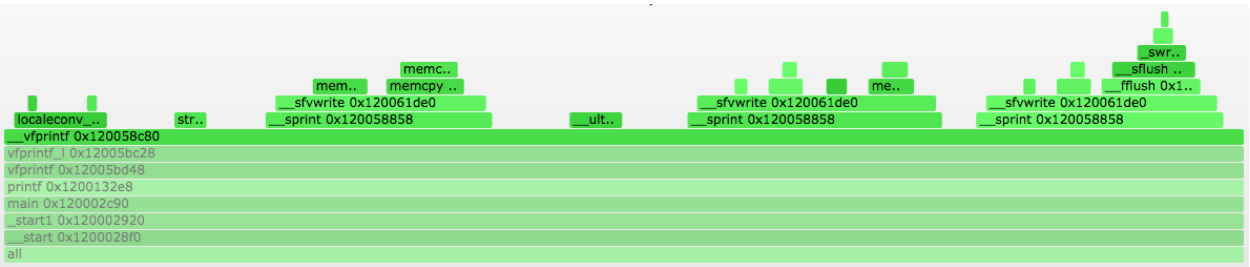

Figure 6: Flame chart of `__vfprintf()` execution, post buffer allocation

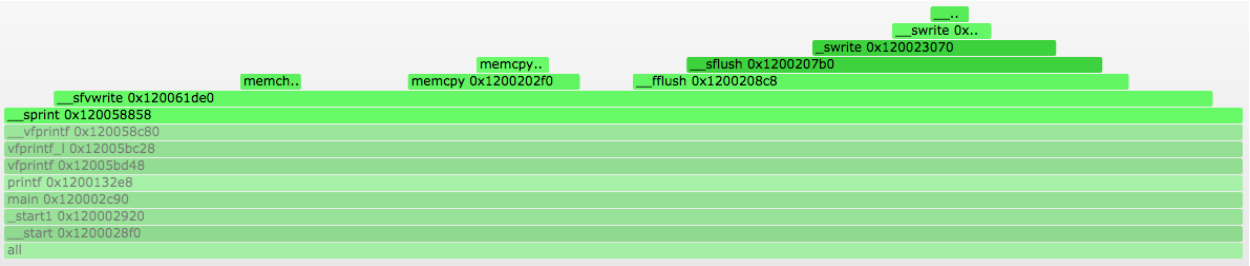

Figure 7: Flame chart of `__sprint()` execution

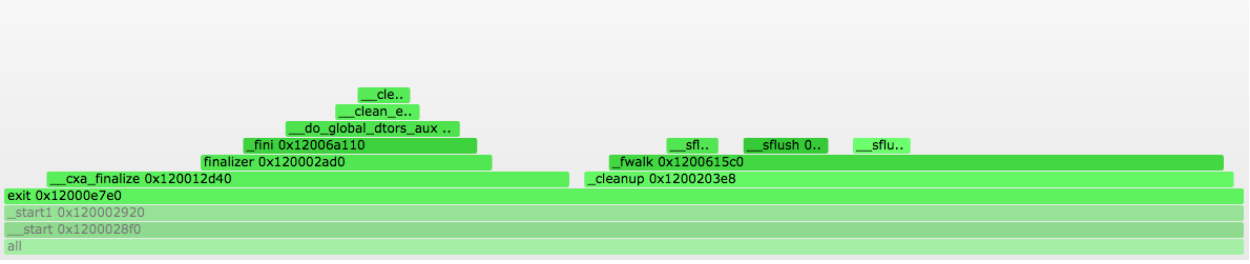

Figure 8: Flame chart of `exit()` execution

largely proceeds as before. The primary exception is that calls to symbols in another library result in symbol lookups on the first trip through. These lookups are performed by the `_mips_rtld_bind()` function, which locates the function in question, updates the pointer in the Global Offset Table (GOT), and calls the function. These lookups are performed on first use because they are quite expensive (about ninety-thousand userspace instructions on FreeBSD MIPS when calling into `libc`), so it is important to avoid performing them on functions that are linked but not actually called by a given execution of the program.

The way we call `_mips_rtld_bind()` is interesting. Each undefined function (that is to say, each function whose implementation is expected to be found in another library) has an entry in the global offset table. On MIPS, this initially points to a tiny bit of stub code in the `.MIPS.stubs` section – which loads the offset of the GOT entry and the original return address into a temporary register and calls the `_rtld_bind_start()` function. The stub for `atexit()` in our dynamic `helloworld` program is:

```
ld      t9,-32752(gp)
move    t3,ra
jalr    t9
daddiu  t8,zero,9
```

`_rtld_bind_start()` then sets up a stack frame saving the usual values including all argument registers, but with the original return address rather than the address of the stub stored and calls `_mips_rtld_bind()`. `_mips_rtld_bind()` performs some locking, looks up the location of this function in the GOT, and locates teh symbol address by calling the run-time linker function `find_sysdef()`. When the symbol is is returned, its address is stored in the GOT (so we don't need to call the stub code again), and we return to `_rtld_bind_start()`. The remainder of `_rtld_bind_start()` restores the argument registers with which it was called, as well as the return address of the stub code. It then makes a tail call to the newly discovered address of the function.

5 Conclusions

All the pieces required to make a simple `helloworld` program work require considerable engineering efforts. A few things are influenced by the need to support legacy code. However, the vast majority of the half-megabyte size of a static `helloworld` is there currently for good reasons. Indeed, the engineering efforts form the basis for much more complex systems that help run our daily lives.

Were you really afraid to ask? Or glad that you did?

6 Further reading

If you would like to know more about the underpinnings of `helloworld` and other C programs, a number of resources are available. The classic book *Linkers and Loaders*[2] covers linking in considerable detail. It's showing its age, but still worthwhile. For a more modern view, Ian Lance Taylor has a series of blog posts on linkers beginning with . For aspects of startup files, the *GCC Internals Manual* provides some useful coverage. The layout of the stack prior to return from `execve()` is architecture dependent, but for ELF systems derives from the System V i386 ABI[3] designed by SCO.

7 Acknowledgements

I thank my colleagues Jonathan Anderson, Ross Anderson, David Brazdil, Ruslan Bukin, Gregory Chadwick, David Chisnall, Nirav Dave, Khilan Gudka, Alexandre Joannou, Bob Laddaga, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Simon W. Moore, Peter G. Neumann, Robert Norton Alex Richardson, Michael Roe, Colin Rothwell, Howie Shrobe, Stacey Son, Munraj Vadera, Stu Wagner, Robert N. M. Watson, Jonathan Woodruff, Bjoern Zeeb for their feedback and assistance. We would also like to thank Şerban Constantinescu for providing traces of Android's JNI usage and measuring CheckJNI overhead. This work is part of the CTSRD and MRC2 projects sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contracts FA8750-10-C-0237 and

FA8750-11-C-0249. The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government.

References

[1] KERNIGHAN, B. W. *The C Programming Language*, 2nd ed. Prentice Hall Professional Technical Reference, 1988.

[2] LEVINE, J. R. *Linkers and Loaders*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[3] THE SANTA CRUZ OPERATION, INC. System V application binary interface, Intel386™ architecture processor supplement (fourth edition). Tech. rep., 1996.