General Information

Name: Tyler Baxter

Email: sterspark@gmail.com

Phone: +1 936 404 5596

IM/IRC: agge (IRC)

Availability: *Anytime*, 6 PM UTC - 5 AM UTC

Working hours:

- Monday-Friday, 6 PM UTC - 2 AM UTC (1 hour lunch)
- 35 hours a week
- *Small size project* - 9 weeks long, 90 hours (total)

Available for contact after work hours and on weekends

Other obligations:

- Summer course (Data Structures) at my college. Course is asynchronous online with two in-person exam days (July 8th and July 29th)
- Dentist appointment June 7th. Root canal (TBA)
- No other obligations expected

Biography

I love UNIX history. FreeBSD is one of the last remaining FREE BSD implementations. My previous experience with FreeBSD is with a personal server. I currently have FreeBSD installed on a Linux QEMU/KVM virtual machine. My major driving passion behind FreeBSD is the permissive BSD license, as opposed to the more restrictive GPL. FreeBSD is a more cohesive "vision" than Linux. System administration tasks are straightforward, the handbook is complete, and utilities are homegrown. Compiling from source is first-class (with ports). Although I am a first-time contributor, I enjoy my experience the more I interact with FreeBSD. Google Summer of Code will be an amazing opportunity to become a part of FreeBSD and develop lasting relationships with the community.

My educational goals are to be a systems programmer. FreeBSD, naturally, aligns with my motivations, and will continue to align with my motivations in the future. My curriculum has been in C++. A notable course I have taken is Computer Architecture (*Computer Organization and Design*, Patterson and Hennessy). Outside of education, I have continued to pursue systems programming, being proficient in C and shell scripting. I'm weaker in shell scripting, so I hope to improve and have a powerful "glue" language for my C/C++ background, the ideal systems programming stack. Extracurricular projects I have are

`Game engine (C++)`
: *Advanced OOP (game entities), project organization (multiple dir, include, and src), CMake, Doxygen documentation*

`Linux lock screen (C)`
: *D-Bus low-level C API and Wayland protocol*

`System utility experiments (C, bash, Perl)`
: *HTTPS server, listener daemons, etc.*

`Functional programming (Lisp-dialect Racket)`
: *Education and frequent personal use*

I am comfortable referencing man pages in systems programming, and am familiar with the overall procedure. I am used to git revision control and use it regularly for my personal and educational projects. Notable programming experience with FreeBSD is recently diving into `bsdinstaller`, utilizing `rc(8)`, `services(8)` , `sysrc(8)`, `bsddialog(1)`.

My experience with Lua is with Neovim configuration and preparation for this project. Difficulties I will face in this project are using Lua in such a complex way and working one step below what man pages can provide. I have programmed complete programs in MIPS assembly and had some exposure to amd64. I have the preliminary low-level knowledge for this project, though that will be an area I grow. At the end of this project I will be very proficient in advanced systems scripting and be a confident systems programmer.

Possible Mentor: TBA

Project Information

Project Title

Refactor syscall creation script

Project Description

The current FreeBSD system call creation script, `sys/tools/makesyscalls.lua`, was implemented by Kyle Evans and iterated on by Brooks Davis. Its purpose is to streamline the introduction of system calls into the FreeBSD kernel. *So you want to add a syscall?* by Brooks Davis establishes the flow of system call creation

Add the new system call entry and its interface into `syscalls.master`

`make sysent` calls `makesyscalls.lua` with `syscalls.master` and `syscalls.conf` as arguments

`makesyscalls.lua` handles much of the dirty work, streamlining the process

The default outputs files are

`syscalls.c`	`syscalls.h`	`syscalls.mk`
`init_sysent.c`	`systrace_args.c`	`sysproto.h`

`makesyscalls.lua` is a transliteration of FreeBSD's original `makesyscalls.awk` script from awk to Lua. It's a monolithic script that has kept much of the procedural-oriented awk code. It does not take full advantage of modern features of Lua (object-oriented design) and has been difficult to add additional features to. FreeBSD system call creation will be further improved by a complete refactor of `makesyscalls.lua` to have easily extensible objects and dynamically-called modules. The goal is to strip the monolithic script into an easily extendable interface. It will utilize classes to give extensible methods to objects, mainly utilizing the facade and command design patterns, to give a more "bite-sized" interface to the previously procedural code. This will allow easier maintenance of the Lua tables and procedures, and a better platform to add more. Lua modules will provide better namespacing of globals, dynamic generation of output files, and more modules with different tasks may be easily added. After refactoring, the previous functionality of the original script can be achieved simply by calling the necessary modules.

`makesyscalls.lua` will be stripped back to its core functionality, providing much of its implementation as a class interface

> *e.g.,* *`FreeBSDSyscall:new() syscall:compat_level() scarg:arg()`*

Globals namespaced in modules

> *e.g.,* *`config.process(fh), config.get(), config.is_modified()`*

Modules for dynamic generation of output files

> *e.g.,* *`init_sysent.lua, sysproto_h.lua, systrace_args.lua`*

Calling `freebsd.lua` will produce the functionality of the original script. By taking advantage of modern Lua features and object-oriented design, however, it can be easily extended, called dynamically, or serve as the basis for future scripts.

The benefit to FreeBSD is further streamlining in the creation of system calls, a more maintainable interface, and a strong foundation to build upon in the development of future system call creation tools. There is a clear intent to expand on the work of Kyle Evans and Brooks Davis, and my refactor will address that intent and provide an extensible interface to do so. It will further demystify the process of system call creation and allow others to more easily contribute. *My refactor will enable FreeBSD developers to more easily create development tools and improve the FreeBSD system call creation process.*

Warner Losh has done preliminary work on refactoring `makesyscalls.lua`, which will serve as the basis for my refactor. It is unfinished and does not incorporate Brooks Davis' recent commits. A successful project will be finishing the pre-established work, incorporating recent commits, and the previously stated design outcomes.

Warner Losh's preliminary work

`util.lua` module	*for utility functions*

`config.lua` module	*for processing config file*
`init_sysent.lua` module	*generates `init_sysent.c` entry* ***(incomplete)***
`syscall_h.lua` module	*generates `syscall.h` entry*
`syscall_mk.lua` module	*generates `sycall.mk` entry*
`class syscall`	*encapsulates the processing of a syscall, produces a generic syscall interface*
`class FreeBSDSyscall`	*preprocessed FreeBSD syscall object*
`class scarg`	***(unimplemented)***

Brooks Davis' recent commits

libsys: don't try to expose yield
lib{c,sys}: expose _getlogin consistently
makesyscalls: generate private syscall symbols
makesyscalls: add COMPAT14 support
makesyscalls: don't make syscall.mk by default

There is still much work to be done and critical design choices to be made (e.g., *is it better to have local write procedures or a class interface?*). Due to the nature of refactoring, changes may occur during the project.

FINAL IMPLEMENTATION

`freebsd.lua (core)`	`util.lua`	`config.lua`
`syscall_h.lua` `sysproto_h.lua`	`init_sysent.lua` `systrace_args.lua`	`syscall_mk.lua`
`class FreeBSDSyscall`	`class syscall`	`class scarg`

Deliverables

System call creation will work as before

`makesyscalls.lua` *is refactored into core, modules, and classes*

System call creation library is easily extensible
(It should provide a basis for future system call creation scripts)

Well-documented

  "`bsd_foo` will be generated"
  How to opt-out of complex generation

GSOC MIDTERM EVALUATION: *Implement `class scarg`*

MILESTONE 1: *Implement `class scarg`*

In `makesyscalls.lua`, `process_args()`

`bool changes_abi` *is a flag for ABI changes*

  `bool check_abi_changes()` *checks for ABI changes*

`strip_arg_annotations()` *removes Microsoft SAL and leaves just type*

Replaces argument types with respective ABI config types

  `isptrtype()` *for pointer condition (checks for `*` or `caddr_t`)*
  `isptrarraytype()` *for array pointer condition (checks for `**` or `const[]*`)*
  `is64bittype()` *for 64-bit type condition*

Assigns bit padding flag to signed int

  For padding expansion macros - architecture and endianness

Adds processed arguments (type and name) to a table
Returns table and `changes_abi` flag

`process_syscall_def = λ`

Calls `handle_compat()`, `handle_noncompat()`, etc.

Defines and processes syscall return value (0 for success, -1 for error and `errno` set)

Flags, function name, function arguments, etc.

File output is to `sysproto.h` and `systrace_args.c`

Everything is very procedural and very coupled
I will uncouple and implement a high cohesion interface
-> Function arguments and return value

Much of the heavy lifting will be done in private methods, refactoring individual functionality into methods. More methods can easily be added, achieving the goal of extensibility. The final interface will be `scarg:arg()` and `scarg:ret()`, a high cohesion interface for future scripts.

GSOC FINAL EVALUATION:
`makesyscalls.lua` refactored into a library of modules and classes

MILESTONE 2:
Finish implementation of `init_sysent.lua` module

MILESTONE 3:
Dynamically generate output files (as a side-effect of a more extensible interface)

The solution is separate module subroutines of previous `loop write_line(xxx)` procedure of producing auto-generated files. Common methods will be put into interfaces

Auto-generation of `init_sysent.c`, `systrace_args.c`, `sysproto.h` has common procedures

Currently, `handle_compat(), handle_noncompat(), handle_obsol(), handle_reserved()` do the bulk of the work

Available data must be `sysnum, thr_flag, flags, sysflags, rettype, auditev, funcname, funcalias, funcargs, argalias, syscallret`

Data will be provided by `class syscall` or `class scarg`

Data for `handle_xxx()` will be refactored into respective classes

-> State pattern will give objects polymorphism, providing cohesive interface

`freebsd.lua` will be able to dynamically generate each output file via modules

-> Implementations will be as a module, or as an interface of a class

Other examples of refactoring...

`if configfile ~= nil then ...`

Will be a function of the `config.lua` module. It doesn't need to be called at every entry point (redundant) and is more declarative and reusable being namespaced (e.g., `config.get()`)

`abi_changes()`

Is used by both `sysproto.h` and `systrace_args.c`. It will be publicly available

Test Plan

Due to the nature of refactoring, it's difficult to have iterative live testing. Local unit testing will be necessary until midterm evaluation, but after the implementation of classes, it's possible to do live regression testing on the file output of individual modules.

It is important to immediately migrate Brooks Davis' commits into Warner Losh's preliminary work, so the refactor can be tested against FreeBSD CURRENT. That will be my starting point.

Lua is an interpreted scripting language, meaning two great things for testing: rapid iteration and REPL. There are a lot of moving pieces to `makesyscall.lua`. `pcall()` and `assert()` may be used liberally. Methods will be built up piecemeal, unit testing correct behavior and outputs of the methods. `makesyscalls.lua` currently functions correctly, so certain assumptions of Lua chunks will be made (e.g., `isptrtype()` will be assumed to work when refactored), though unit testing will still be employed to build up methods.

Being a refactor, regression testing will be the primary testing procedure.

Unit testing will confirm…

> `class scarg` performs (refactored) `process_args()` and `process_syscall_def = λ` as expected
>
> `scarg:arg` matches `return` of `process_args()`
>
> `scarg:ret` matches `syscallret` of `process_syscall_def = λ`

Regression testing will occur…

After implementation of `class scarg`	Generation of `syscall.h` will be regression tested against previous generation
After implementation of `init_sysent.lua` module	Generation of `init_sysent.c` will be regression tested against previous generation
After implementation of `sysproto_h.lua` module	Generation of `sysproto.h` will be regression tested against previous generation
After implementation of `systrace_args.lua` module	Generation of `systrace_args.c` will be regression tested against previous generation

Due to refactoring limiting testing, an extended period of live regression testing will occur after my refactor has been completed. I will patch as necessary, until system call creation works as before.

Project Schedule

Week 1

`Goal:` Migrate Brooks Davis' commits. Separate auxiliary procedures out of `process_args()`

`Result:` *Brooks Davis' commits migrated. A solution to `bool changes_abi, bool check_abi_changes(), isptrtype(), isarrayptrtype(), is64bittype()`*

Week 2

`FRIDAY JUNE 7th OFF, DENTIST APPOINTMENT`

`Goal:` Reconstruct `process_args()` into private methods of `class scarg`

`Result:` *Arguments processed and ready to provide public interface `scarg:arg()`*

Week 3

`Goal:` Refactor to achieve accessible `scarg:arg()`

`Result:` *`scarg:arg()` returns equivalent of `return funcargs` from `proccess_args()`. Solution to `bool changes_abi`*

Week 4

`Goal:` Refactor to achieve accessible `scarg:ret()`

`Result:` *`scarg:ret()` returns equivalent of `local syscallret` from `process_syscall_def = λ`*

MILESTONE 1: Implement `class scarg` (COMPLETED)

Week 5

GSOC MIDTERM EVALUATION: MILESTONE 1 (COMPLETED)

Goal: Tidy up miscellaneous functions, implementing in modules or classes as necessary

Result: *Global functions properly namespaced, or in a class interface*

Week 6

Goal: Make parameters of `handle_xxx()` accessible to `init_sysent.lua` module

Result: `init_sysent.lua` module outputs `init_sysent.c` entry the same as previous `makesyscalls.lua`

MILESTONE 1: Finish implementation of `init_sysent.lua` module (COMPLETED)

Week 7

MONDAY JULY 8th OFF, COLLEGE MIDTERM

Goal: Refactor generation of `sysproto.h` into `sysproto_h.lua` module

Result: `sysproto_h.lua` module outputs `sysproto.h` entry the same as previous `makesyscalls.lua`

Week 8

Goal: Refactor generation of `systrace_args.c` into `systrace_args.lua` module

Result: `systrace_args.lua` module outputs `systrace_args.c` entry the same as previous `makesyscalls.lua`

MILESTONE 3: Dynamically generate output files

(COMPLETED)

Week 9

Makeup days off

Goal: Live regression testing, patch as necessary. Provide documentation

Result: *All deliverables provided*

GSOC FINAL EVALUATION: MILESTONE 2 & 3 COMPLETED

`makesyscalls.lua` **refactored into a library of modules and classes**