CMSC430 Final Project

Spite: Sockets & Syscalls

Hana Chitsaz, Lincoln Doney, Joseph Lewis

Introduction

The name of our language is Spite: Sockets & Syscalls. It contains a range of features, primarily with the goal of having more intractability with the system. Instructions to test the code are in the README of the repository.

Project Selection

When implementing our language, we had one overall metric of success: can we implement an HTTP web server. To allow for an HTTP server to be run on Linux Systems, we implemented basic C-like I/O, and C-like sockets. We also implemented a notion of libraries and imports in our language in order to construct an HTTP library which we could interface with. The languages which we worked with in class were not expressive enough to interface with the system at all, and were also very limited in program size due to requiring every program to be in one file, so these features were all necessary to reach our goal.

High-level Implementation

We decided to fork our language from the most recent language reviewed in class, Loot. For our I/O and sockets, because a86 does not have any notion of system calls or sockets, we had to implement these as C external calls. Almost all of the file/socket-related primitives are just calls to C functions which call the C version of the syscall. Libraries are also fairly C-like, in that they effectively[1] paste all of the contents of the other program into the currently working program. In order to help write a standard library of some kind, we also ported over projects 3, 4, and 5 in order to have more expressive programs. We wrote a few libraries: strings, lists, and util. Finally, we combined all of these features together, and created a simple HTTP web server.

Implementation

Files/Sockets

Files provide the program with capabilities to interface directly with the operating system's files. Sockets provide the programs the ability to open and close network sockets and communicate with other devices. In implementing files and sockets, we had decided to follow the C-like method of using them. This being, that opening one with **open** returns and integer file descriptor which can

[1] They actually do a lot more than this because of function name clashes, the way defines are handled, and circular dependencies, but this suffices for a simple explanation. It will be explained in-depth later.

be interfaced with later using **read**, **write**, and some other useful system calls for control such as **close** and **accept**.

To implement files and sockets, new primitives were introduced which effectively call the system calls in C, because a86 does not have any syscall functionality built in. These were all implemented in io.c, and are all labeled "spite_{syscall-name}". The main difficulty in performing this translation was that C strings and spite strings are implemented in memory very differently. C strings are composed of 8 bit (1 byte) characters, while spite strings are composed of 32 bit wide characters. So, a translation stage needed to be implemented, which is performed in "stoc_str" and "ctos_str", which performed spite to C string and C to spite string conversion respectively. With these in place, writing the system calls is just a matter of wrapping the C functions with the translation overtop. These are all called as any extern call is, using the Call directive in a86.

Sockets and Files used the same system calls where appropriate. For example, since the underlying Linux write syscall works on both sockets and files (any file descriptor), so does the write call in our language. Sockets, since they operate on network data which may not be immediately available, we created another function called on-message which can be thought of as an "on-data" callback like those used in Javascript API calls. Each time the socket reads any available data, the lambda passed in as an argument is called with the data. The on-message function only returns when the underlying socket is closed or there is an error.

The other obstacle was in implementing the on-message function. Since this function takes a lambda as an argument that is called by C whenever data is received, we needed some way to call Lambdas from C. Since the application process for a lambda is quite complex and involves manipulating the stack, we give C an address to "wrapper" assembly code that simulates an application. This wrapper code sets the stack up so the lambda can execute properly. This was initially quite a challenge, since we needed to think in-depth about how function applications use the stack and how C uses the stack. In addition, since the lambda provided can also call C functions, we needed to save callee-saved registers, including r15, to avoid bugs.

Example Code

echo_server.rkt

```
#lang spite
(define (echo serv)
  (let ([peer (accept serv)])
    (begin (write "Accepted peer\n")
           (on-message peer (lambda (msg) (write peer msg)))
           (write "Peer disconnected\n")
           (echo serv))))
(let ([serv (listen 8080)])
```

```
   (begin (write "Listening on port 8080\n")
          (echo serv)))
```

echo_client.rkt

```
#lang spite
(include "lib/strings.rkt")
(include "lib/lists.rkt")
(include "lib/util.rkt")
(define (echo sock n)
  (let ((user-in (read n)))
    (if (util:equal? (lists:take-only (strings:str->char-list user-in) 2) (lists:list #\:
#\q))
      (println "Done!")
      (begin (println user-in) (write sock user-in) (read sock n) (echo sock n)))))
(let ([sock (open-sock "127.0.0.1" 8080)])
   (begin (println "Type :q to quit") (echo sock 15) (close sock)))
```

Libraries

Libraries provide the programs with the ability to reference previously written programs or even build up complex libraries with predefined functions for the program to use. The main decision here was if we were going to make our libraries more C like, which involves just pasting in the contents of the other file, or more Java like, where there are complex namespaces and library structures. We decided to do sort of a combination of the two, but leaning much more towards C like. The main reason for this is that C like dependency structures are simple and easy to understand, and don't require a significant overhaul of the parser. But, we also wrote in some basic namespacing features to prevent name clashes between imported libraries.

Libraries were almost exclusively implemented in the parser, mostly as a post-parsing step. First, all of the **include** directives were all read and interpreted, along with any meta-data about them such as the **as** primitive. A program now has a list of *Includes* which describe the dependencies of the file. There are two primary reasons why this was done as opposed to parsing the dependencies as they were read: first, to fix infinite looping on circular dependencies, and second, to future-proof the library feature so that more complex dependency structures could be implemented later on.

After parsing is concluded, all of the files which are referenced are parsed. Every definition made in that file is then added to the main list of definitions as "lib-name:defn-name". Any executable program associated with the file is ignored. If the file which is being parsed also contains includes, this whole process is recursively repeated for that file.

Along with the implementation of the primitives, some basic libraries were introduced to debug and exhibit the capabilities of the libraries feature. They were also used to assist in constructing the HTTP web server: the goal of the project as a whole. To assist in making the libraries, we also ported over

projects 3,4, and parts of 5, in order to make the language slightly more expressive[2]. Each library included some functions which could be used to simplify the job of the user, like a **map** function in lists.rkt or **str-eq?** in strings.rkt.

Example Code

mylib.rkt

```
#lang spite
(define (list . xs) xs)
```

mylib2.rkt

```
#lang spite
(define (first lst) (car lst))
```

main.rkt

```
#lang spite
(include "mylib.rkt")
(as "lib" (include "mylib2.rkt"))
(define (f x) (add1 x))
(f (lib:first (mylib:list 1 2 3))) // Outputs 2
```

HTTP Library/Web Server

In order to implement the HTTP library, a few functions were implemented to make the job of running an HTTP server easier. This was the last step in our project, as it simply was a composition of functions and features we had already written. The main entrypoint is **http-serv-file**, which listens to a client and responds to requests to / as the argument to file. If the user passes anything other than a GET request to /, it returns a 404. There are also functions **get-path**, **get-method**, which return the path and the method of the request. Finally, there are **is-get?** and **is-index?** which return true if the request is a GET request, and true if the path is / respectively.

Then, we used the http library in a simple webserver program, which just uses the http library to serve a single file, index.html, listening on port 8080.

web_server.rkt

```
#lang spite
(include "lib/http.rkt")
(begin (println "Starting web server on port 8080")
       (let ([serv (listen 8080)])
            (http:http-serv-file serv "index.html")))
```

[2] The main reason was to introduce rest arguments, as it helped significantly with implementing list functions.

Testing

Because most of the language features explicitly require system-wide side effects, thorough unit tests were difficult to implement.
In testing libraries, we wrote a few, potentially useful, libraries, which cross-depended on eachother where necessary. The functions were also made arbitrarily complex, to confirm that imports worked properly and were not mis-copied over in the post-parsing stage. Many of the functions implemented in lib/lists.rkt and lib/math.rkt were mostly used for testing. Because this was primarily in the parser, more sophisticated tests weren't really feasible.

For files, we just created a program which goes through a file and prints it to the console, and tested it with edge case files. For sockets, we tested it by first implementing clients and communicating with Google (8.8.8.8), then by implementing servers and communicating with netcat, and then by implementing a two-way echo server. Writing more sophisticated tests was challenging, and thus we mostly just did implementation tests.

The final test, which was the goal of the project, was to ensure that the HTTP server was able to serve a file, and a client would be able to receive it on the other end. The file we serve, index.html, simply has the contents

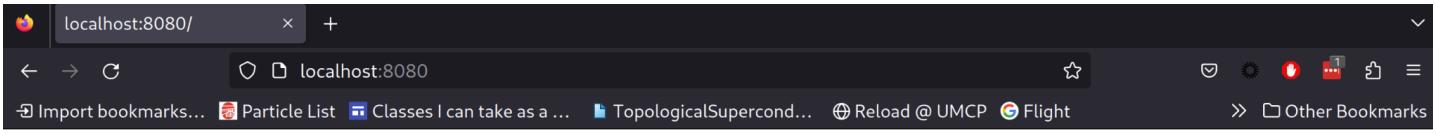
index.html

```
<h1>I'm alive! Hello, from spite!</h1>
```

Upon running the webserver and then navigating in the browser to http://localhost:8080/, we get the following console output:

```
~/C/Spring 23/C/FinalProject/sample main !2 ?3 ❯ ./web_server.run
Starting web server on port 8080
Got request!
```

And the following web page was rendered



I'm alive! Hello, from spite!

So it works! We get a rendered webpage which is served from our code. Of course, the file we served is very simple. However, more complex websites could just as easily be served, as the HTTP server just sends a file.

Conclusion

Overall, the language features seem to work together very well and are theoretically expandable into more complex programs. The programs implemented in this project are all fairly simple compared to the capabilities of the language as a whole.

There is one major limitation with the language. Because of the system nature of our language, the language only works for linux-based systems. In the future, if we were to continue to implement the language, making it work for OSX and Windows is a matter of using preprocessor directives to handle OS differences on the system calls. The other major limitation is in the way our sockets work: only one peer can be served at a time. This is because our language has no notion of threading. Also, implementing more complex string parsing functions to deal with more complex web requests would make our language much more sophisticated and expressive, but this is part of our standard library and not the language itself.

Language Reference

This wasn't part of the project report officially, so consider it distinct from the report. However, we thought it would be useful, so we included it.

File/Socket Operations

Opening a file
- (**open** {string:PATH} {char:OFLAG}) -> file
  - OFLAG is one of: #\r -> read, #\w -> write, #\a -> append

Opening a socket
- (**open-sock** {addr:string} {port:int}) -> socket
  - Opens a TCP socket to a specified address/port pair
  - Addr is only an IP address (for now)
- (**listen** {port:int}) -> socket
  - Opens a listening TCP socket for servers

Closing a file/socket
- (**close** {file/socket:FILE}) -> bool
  - In our language always returns true because false would just result in error and kill the program

Reading/Writing from a file or socket
- (**read** {file/socket:FILE} {int:NUM_CHARS}) -> string
- (**write** {file/socket:FILE} {string:STR}) -> void

Reading/Writing from Stdin/Stdout
- (**read** {int:NUM_CHARS}) -> string
  - reads from stdin (modified read_byte)
- (**write** {string:STR}) -> void
  - writes to stdout (modified write_byte)

Socket Management
- (**accept** {sock:socket} -> socket
  - Gets a new peer socket from a listening socket.
- (**on-message** {sock:socket} {do:lambda (msg:string)}) -> void
  - Repeat method passed as lambda for every message received on socket
- (**closed?** {sock:socket}) -> bool

New types
- File (int): underlying file descriptor that is returned by C's open
  - Associated assert: (assert-file)
- Socket (int): underlying file descriptor for C's sockets
  - Associated assert: (assert-socket)

Libraries

- (**include** file:str)
  - Includes all defines in file and adds them to our list of defines in current position
- (**as** libname:str include:inc)
  - Changes the name of the include inc to libname.

String Library
- (**str->char-list** {string:str}) -> list of chars

    - Takes a string and returns a list of characters that preserve the order of the chars
    - Essentially breaks down the string into a list of individual characters
    - Uses an internal recursive helper function
- (**char-list->str** {list of chars:cs}) -> string
    - Takes a list of characters and returns a new string joining the characters of the list
    - Uses an internal recursive helper function
- (**string-eq?** {string:str1} {string:str2) -> bool
    - Returns #t if str1 and str2 are equal
    - Returns #f (not an error) if str1 or str2 are not strings
    - Uses an internal recursive helper function
- (**contains?** {string:str} {char:c}) ->bool
    - Returns true if the character c is in the string c
    - If given anything that is not a string or char will, it will throw an error.
    - Uses an internal recursive helper function
- (**get-index** {string:str} {char:c})->int
    - Returns the index of the first instance of the char c in string str
    - Uses an internal recursive helper function
    - If the character c does not appear in the string str it will return -1
- (**string-copy** {string:str})->string
    - Returns a new string that is the same length and holds the same characters as string str
    - Uses an internal recursive helper function
- (**substring** {string:str} {int: i}) ->string
    - Returns a new string starting at the index i (inclusive) until the end of the string
    - Uses an internal recursive helper function
- (**str-char** {string:str} {char:c}) -> string
    - Returns a new string that starts with the first occurrence of c in str.
    - Returns a zero-length string if C does not occur in str.
- (**remove-first-char** {string:str})->string
    - Removes the first character of the string str
- (**get-first-chars** {int:n} {string:str})->string
    - Returns a new string with the first n (inclusive) characters of the string str
- (**str-append** {string:str1} {string:str2})->string
    - Returns a new string adding str2 to the end of str1
- (**string-assign** {string:str} {int:i} {char:c}) -> string
    - Primitive that replaces a character at index i with character c in string str

Util Library

- (**equal?** {T: e1} {T: e2}) -> bool
    - General statement of equality for two expressions of the same type

    - If type of e1 does not equal type of e2 it will return #f as well as if the two expressions of the same type are not equal to each other.
- (**veq-eq?** {vector:v1} {vector:v2}) -> bool
    - Checks if two vectors are equal
    - If the two vectors are not equal or they are not of type vector it will return #f
    - Uses an internal recursive helper function
- (**xor** {bool: b1} {bool: b2}) -> bool
    - Computes the xor of two boolean expressions b1 and b2
- (**not** {bool: b}) -> bool
    - Computes the not of the boolean expression b
- (**or** {restof bool: bs}) -> bool
    - N-ary or of boolean expressions
    - returns #t if at least one boolean expression in bs is true
- (**and** {restof bool: bs}) -> bool
    - N-ary and of boolean expressions
    - Returns #t if all boolean expressions in bs are true

HTTP Library

- (**get-method** {string:s}) -> string
    - Gets the method from a HTTP request string
- (**get-path** {string:s}) -> string
    - Gets the path from a HTTP request string
- (**send-file-resp** {socket:peer} {file:f}) -> bool
    - Sends HTTP response containing contents of file f
- (**http-resp** {socket:peer} {file:f}) -> bool
    - Respond to a web request. If OK, send contents of f, otherwise, return 404
- (**http-serv-file** {socket:sock} {file:f}) -> void
    - Accepts socket and awaits message from client

Miscellaneous

- (**throw-error**)
    - Primitive in compile-op0 that jumps to 'raise_error_align
    - A way of throwing an error outside of the compiler files