LUA 1.1 shell scripting project

Summary

This project is to add the ability to use LUA as a shell script much like bash in Linux style systems. Type a script file name and it should run. We decided on version 1.1. While we didn't manage to get all the functionality finished the basics are there.

You can type the script name, and if it has the proper signature it will be handed to the interpreter to run. Without file reading and writing working fully you are very limited in what you can do. It will not be too hard to add that ability with everything else functioning.

Getting it to function required reinventing the wheel but it's very good practice in working in C. It also gives you a huge appreciation of all the work a professional library both does for you and how much work it took to create.

All C std library declarations and description on how they work came from **cppreference.com.** Not all functions have the direct link due to time limitations but they all came from that website. There are many functions with descriptions and direct links. Make sure you look at the C version and not the C++ version. All actual code is written by us.

How well did we follow the design, and if not why not

We actually followed it almost exactly to our amazement. The differences will be shown after a quick history of the work before we made the design document.

Quick history

So right after our proposal was accepted we spent 4 days straight seeing if it was even feasible to get LUA working on Xv6. We first tried 5.3.4 which ended up being over 300KB _lua file, and we included all the LUA files as the user library which heavily bloated all the other files as well. It was obvious this wouldn't work.

The next thing we tried was version 1.1, which was the first official public release. This ended up still well over 60KB max file size for Xv6. We then tried the -Os option which took it down to 116KB or so. This was still too large. On the 4th day of Easter break we had an idea to increase the number of direct blocks. We found out 124 is the largest that you can set it and still have it compile and mkfs.c function. And with this we had 126KB max file size to work with. LUA couldn't do anything since all C std functions it required were simply return NULL; or return 0; while also doing printf(c std name). But it was start. We then focused on getting the command line input part working which calls dostring(cstring). It's easier to get working than file loading for dofile(filename) in lua.c's main func.

We managed to get it to print strings and concatenate them. Math could be done but there was no way to print out a double at that point so it always came back as 0. Internally all numbers are a double in LUA. That was all we had done prior to doing the design document. The string support was added long after the first 4 days though. We just had it compile and run the LUA interpreter which could do nothing.

Differences between design doc and actual implementation

The first difference would be the definition of myfile/FILE. It was changed to

```
//typedefs
typedef struct myfile
{
    int file; //file descriptor
    char name[50]; //path and name
    int is_open; //1 if open, 0 if not
} FILE;
```

This happened when we found out LUA has a static file that it uses for any file operation. Since it only uses 1, it greatly simplified how to properly do the C std way. Since it uses only one file at a time and Xv6 does not call fopen, fclose, and so on we use this to keep track of what LUA is doing.

A second difference is we forgot to mention to change the Makefile to not make all warnings into errors. The line from the design document does not change but it was never clarified that on top of the -O2 to -Os you also remove the `-Werror` option. This is due to some debug code left in LUA and to assist with test code. It's nice for some types of warnings but others like unused code/unused variable it gets in the way of testing.

A possible third is that the pseudo code was not 100% followed. The basic idea was but as the status report showed we had more variables added and adding things to argv is not as single as the pseudo code showed. exec.c has the full version we demoed. Another change is that the header check still happens but only after the LUA header check. This way it confirms that lua, the path and filename for the interpreter is good and an executable. Nothing major changed.

What we learned

We learned how easy pointers are to get wrong. Many of our problems came down to arithmetic issues, pointing the wrong pointer to the wrong thing, dereferencing at the wrong time, and so on. With all the other troubles we ran into we didn't have time to do the printf and fprintf to point to sprintf. This would be even easier to mess up just like trying to get fprintf to point to printf properly. For a while we could not get error messages to print because we gave the wrong memory address to the variadic stack.

We learned that MSVS has their stdin, stdout, and stderr defined as a function. This requires a small change on how LUA gets these. Instead of a static define you need to leave the variables where they are and assign them in the library loading code. Even the debug version of LUA is vastly smaller. If I remember correctly the debug .exe is only 30KB or less.

We learned how easy it is to mess up math in LUA given that we are printing ints when they are doubles internally. if(x == 0) is one example. It may be 0.00001234 which is not 0 even if it prints 0. The fix would be to properly display them as a double but that has it's own pitfalls if you code it with a loop. How many spaces of precision will you print?

We learned how simple LUA 1.1 is compared to 5.3.4, the most recent release. The amount o f std functions that the newer version uses is quite large. This is because of UTF-8 support, locale support, and a larger standard LUA library that comes with it. They also have support to easily build the interpreter and compiler in any system through #defines.

We learned quickly how difficult it is to take a code base that is written in multiple files and try to cram it into a single one. Certain structs like Object, node, and Hash rely on each other. This requires you to have them forward declared along with many others parts. Once it's taken care of it is fixed instantly but it's a big gotcha.

Implementation

exec.c's exec function changes are two parts just like in the status report. This first part is how we detect if the file given is a script or not.

```
char lua_path[] = "lua";
char lua_header[] = "--LUA 1.1";
char lua_test[sizeof(lua_header)];
char* script_path = (char*)0; //set to NULL
int is_lua_script = 0;
//lua header check
if (readi(ip, lua_test, 0, sizeof(lua_header - 1)) == sizeof(lua_header - 1)) //skip reading
a null char ;) this tests to see if the file has at least this many bytes
{
    if (strncmp(lua_header, lua_test, sizeof(lua_header - 1)) == 0) //actually see if it
matches signiture exactly
    {
          cprintf("lua script detected\n");
          is_lua_script = 1;
          script_path = path; //save the old path which is the script
```

```
            path = lua_path; //swap lua programs path where the script path was
            cprintf("script path: %s \n", script_path);
            iunlockput(ip); //unlock script file
            end_op(); //end op on script file

            begin_op(); //repeat steps above but now we are using lua's executable instead :D
            ip = namei(path);
            ilock(ip); //don't need to assign pgdir to 0 because it was done before here
        }
    }
```

Second part below is what happens if a script is detected.

```
if (is_lua_script == 0)  {
//original argument pushing code
}
else  {
      //push lua path name and then the script name
      argc = 0; //push lua path
      sp = (sp - (strlen(lua_path) + 1)) & ~3;
      if (copyout(pgdir, sp, lua_path, strlen(lua_path) + 1) < 0)
          goto bad;
      ustack[3 + argc] = sp;
      //cprintf("argv[argc] %s \n", argv[argc]);

      argc++; //push script name
      sp = (sp - (strlen(script_path) + 1)) & ~3;
      if (copyout(pgdir, sp, script_path, strlen(script_path) + 1) < 0)
          goto bad;
      ustack[3 + argc] = sp;
      //cprintf("argv[argc] %s \n", argv[argc]);
      argc++;
  }
```

Next we will look at the more complicated and troublesome C std library functions that we did implement.

First here is the list from the design doc which has not changed.

```
//additions to xv6
//math
double atof(const char* str);
double sin(double x);
double cos(double x);
double tan(double x);
double asin(double x);
double acos(double x);
double atan(double x);
double ceil(double x);
double floor(double x);
double sqrt(double x);
double pow(double x, double y);

//memory
void* calloc(size_t num, size_t size);
int memcmp(const void* lhs, const void* rhs, size_t count);
void* realloc(void *ptr, size_t new_size);
void* memcpy(void * destination, const void * source, size_t num);

//file
int fgetc(FILE *stream);
FILE *fopen(const char *filename, const char *mode);
int fclose(FILE *stream);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *buffer, const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int ungetc(int character, FILE * stream);
char *fgets(char *str, int count, FILE *stream);
int sscanf(const char * s, const char * format, ...);

//strings
double strtod(const char* str, char** endptr);
int isspace(int c);
int isdigit(int c);
int tolower(int c); //not needed
int isalnum(int c);
char* strcat(char * destination, const char * source);
char* strdup(const char *str1);
int toupper(int c);  //not needed
char* strstr(const char * str1, const char * str2);  //not needed
int isalpha(int c);

//system
int system(const char* command); //not needed
int remove(const char * filename); //not needed
```

Out of the ones we implemented, three stick out as trouble makers.
strtod, realloc, and sprintf(also including printf and fprintf)

strtod was a problem due to how converting a string to a double can take quite a few places after the period. This is much easier to do by following the IEEE #### specification that C chose. We didn't have time to learn that so we just chose to print it as an int. The pointer part was also fun to get working where if they give a non NULL for the second argument you need to put that pointer to the character right after the number.

realloc is a problem due to not fully understanding how malloc and free work. We need to read to specification and the completely comment-less functions used in Xv6. While not impossible, it is time consuming just to get one function to work. We need to understand these two to be able to properly write realloc to know how much the previous pointer was allocated and only read that many btyes. Ours will copy past the originals allocated amount since we don't know how large it was.

```
void* realloc(void *ptr, size_t new_size)
{
if (ptr)
    {
        printf("realloc new_size: %d \n", new_size);
        char* src = ptr;
        char* dest = (char*)malloc(new_size);
        int n;
        for (n = 0; n < new_size; n++) //not proper but hopefully works in most cases.
Unsure how we can get the old pointer's size to do proper copy. if String it's easy, if it's
not we don't know where say an int or float stops/begins
        {
            dest[n] = src[n];
        }
        dest[new_size - 1] = '\0'; //make sure the last thing is a null char
        free(src); //free the old memory and point both to nothing aka NULL
        ptr = NULL;
        src = NULL;
        return (void*)dest;
    }
    else //if null it acts like malloc(new_size)
    {
        return malloc(new_size);
    }
}
```

sprintf along with fprintf and printf all should work in tandem. Our idea was to implement fprintf and printf to call sprintf to get a string and then print that/write it to a file. We ran out of time so fprintf only calls printf and printf and sprintf have their own code paths. Not copying all three here due to how large it will be. sprintf is used heavily for all string operations and even the function table in LUA so we had to add support for %10.10s and others documented in scott.c. Check out ulib.c to see the actual code for all three of these functions.

Another problem is talked about in the what we learned section dealing with how floats are promoted to double size in Variadic functions.