This design document covers the following three topics in this order
1- Modifications to XV6 to get the LUA 1.1 interpreter running
2- How to get the LUA interpreter working(these are separate) This covers the C functions you must write
3- XV6 modifications to allow a shell script to run(exec.c changes)

We used LUA version 1.1 in our implementation. It is both smaller in file size and file count, and uses a smaller selection of the C standard library. It also does not use or support C99 features, unlike versions after 5.0.

At the time of writing this document we have the basic string functionality working on Xv6 with no exec.c modifications. You can print strings, concatenate them, and use variables. Tables have not been tested yet. You can do math, but the functions needed to print out floats/doubles has not been completed yet. Thus you can't see what the result is.

-=Modifications to XV6 to get LUA 1.1 interpreter running=-

“Why do we need to do modifications before we can even get the interpreter running?” you may ask. Well, with a halfway implemented C library that only allows printing strings and concatenating, LUA is eating up 116KB. With these modifications you can squeeze a 126KB file into the system. I also changed printf() to match the standard library definition instead of the XV6 one. This makes you modify all of the user programs as well.

Makefile
from

```
CFLAGS = -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD
-ggdb -m32 -Werror -fno-omit-frame-pointer
```

to

```
CFLAGS = -fno-pic -static -fno-builtin -fno-strict-aliasing -Os -Wall -MD
-ggdb -m32 -fno-omit-frame-pointer
```

Without this your _lua file will not fit anyway, shape, or form. This switches from general speed and size optimizations to focusing on size optimizations only. Anything that increases code size is not done. The only other way to make it fit will be to use double/triple/quad indirect links. But this is a much larger change than just adding more direct nodes. Now we have 252, 512 byte blocks. This is exactly 126KB.

param.h
from

```
#define FSSIZE       1000
```

to

```
#define FSSIZE       4000
```

fs.h
Used to be 11. This gives us a lot more space to work with. It's also as high as the value can go and still compile and function correctly. Makes an Inode take up exactly 512 bytes, the size of our sectors.

```
#define NDIRECT 124
```

printf changes(just remove the int fd first parameter) regular C does not have that
user.h

```
void printf(const char* format, ...);
```

printf.c

```
void
printf(const char *fmt, ...)
{
```

further down

```
int fd = (int)stdout;   //~line 46 can leave this comment out
```

-= How to get LUA interpreter working=-

First here is a list of C standard library functions you will need to write.
Also included are the data structures and defines you need to add
all of the prototypes are from cppreference.com
one example is http://en.cppreference.com/w/c/io/fscanf

user.h

```
//defines
#define NULL (void*)0
#define size_t uint
#define EOF -1
#define stdin (FILE*)0
#define stdout (FILE*)1
#define stderr (FILE*)1

//typedefs
typedef struct myfile
{
    int file;
} FILE;

//additions to xv6
//math
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 atof(const char* str);
double strtod(const char* str, char** endptr);
int isspace(int c);
int isdigit(int c);
int tolower(int c);
int isalnum(int c);
char* strcat(char* destination, const char* source);
char* strdup(const char* str1);
int toupper(int c);
char* strstr(const char* str1, const char* str2);
int isalpha(int c);

//system
int system(const char* command);
int remove(const char* filename);
```

Not everything here must be fully implemented to get it running. But it must all be defined. If you don't care to support the LUA math library, for example, you can skip implementing all of the sin/cos/so on functions. Make sure all functions return 0 or return NULL. The interpreter will not actually function of course, but it will compile.

Also, to get the interactive mode interpreter working you need to make changes to how the gets() loop functions.

from

```
if (argc < 2)
 {
   char buffer[2048];
   while (gets(buffer) != 0)
   {
       lua_dostring(buffer);
   }
 }
```

to

```
if (argc < 2)
 {
     printf("start typing commands. seperate with ; once a newline is hit all commands
execute together.\n");

   char buffer[200];
   do
   {
       memset(buffer, 0, sizeof(buffer));
       gets(buffer, sizeof(buffer));
       lua_dostring(buffer);
   } while (1);
 }
```

This gets the job done.

Last but not least you need to combine all 10 files into lua.c to make it easier to compile. This also keeps the user library from becoming too bloated. You will need to forward declare the data structures and some functions in order for it to compile. Move all #defines to the top as well. Headers before C files and so on. It's around 4.400 lines of code combined.

-=XV6 modifications to allow a shell script to run=-

We will modify exec.c to look for a pre defined comment as the first line of the LUA script. This will be the following

--LUA 1.1

-- is a comment in LUA. https://www.lua.org/pil/1.3.html
This will designate if it's a run-able script straight from the shell vs a normal script. This also separates it from a normal executable which have their own signature.

exec.c
```
// Check ELF header
  if(readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
    goto bad;
  if(elf.magic != ELF_MAGIC)
    goto bad;
```

This part checks if the first part of a given file has the proper header. So we will check if it has the LUA header first, and if not we continue on as normal. If it matches our new header then we will instead load the LUA interpreter and pass it the argument of the original filename. We only read the inode direct/indirect links and don't modify them. So no corruption can occur.

So
-readi before the first if and check for lua script first and put the result into int result
```
if(result != sizeof(elf))
```
-if it's not let it continue like normal
-if we match the lua signiture, then we close and release the file lock on the current one
-next aquire the lock and open up the file lua, our interpreter. set that as the current filename and do a readi on that and let exec continue like normal until below
-the above set of ifs will be placed in an outer if block of
if(!lua_script)
```
{ // Check ELF header
  if(readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
    goto bad;
  if(elf.magic != ELF_MAGIC)
    goto bad;
}
```

exec.c
```
// Push argument strings, prepare rest of stack in ustack.
  for(argc = 0; argv[argc]; argc++) {
    if(argc >= MAXARG)
      goto bad;
    sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
    if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
      goto bad;
    ustack[3+argc] = sp;
  }
```

We will push the original filename and no other arguments before getting to this part. increment . So the argv would be "lua filename". Inside the interpreter it will see > 1 argument and then read it as a file and run the script from there.

So, just above the argument pushing
```
char* lua_string = "lua";
if(lua_script)
{
 argv[0] = &lua_string;
```

```
  argv[1] = &filename;
}
```

These get copied out so assigning it to a stack's address will be fine. After they get copied out they no longer point to the stack that we are using. Then the exec function continues as normal and should load our interpreter instead and then the interpreter will get the 2nd arguement and run that script.