Paul Kummer

Dr. Hanku Lee

Operating Systems CSIS 430-01

09/29/2021

Lab 04: Intro to XV6

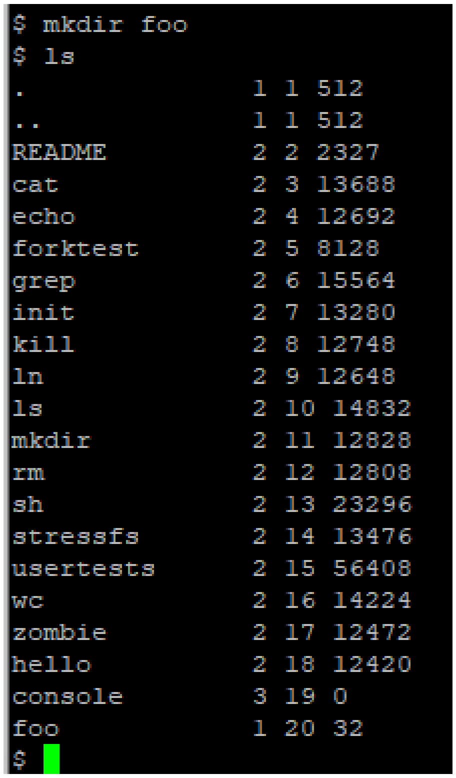
Purpose of Lab

This lab's purpose was to introduce the students to the Unix distribution of xv6. This version is intended to be an educational tool for students to learn about operating systems and how to manipulate them. The focus of this lab was to install xv6 in Smaug so that students could run the OS and create their own command called "hello". After changing the OS's files of Makefile, syscall.c, syscall.h, sysproc.c, user.h, and usys.S and adding hello.c, the student can recompile the OS with the new OS command called "hello". This will then print out "Hello World" to the terminal window when the user types "hello" and hit enter.

Examples

Testing commands

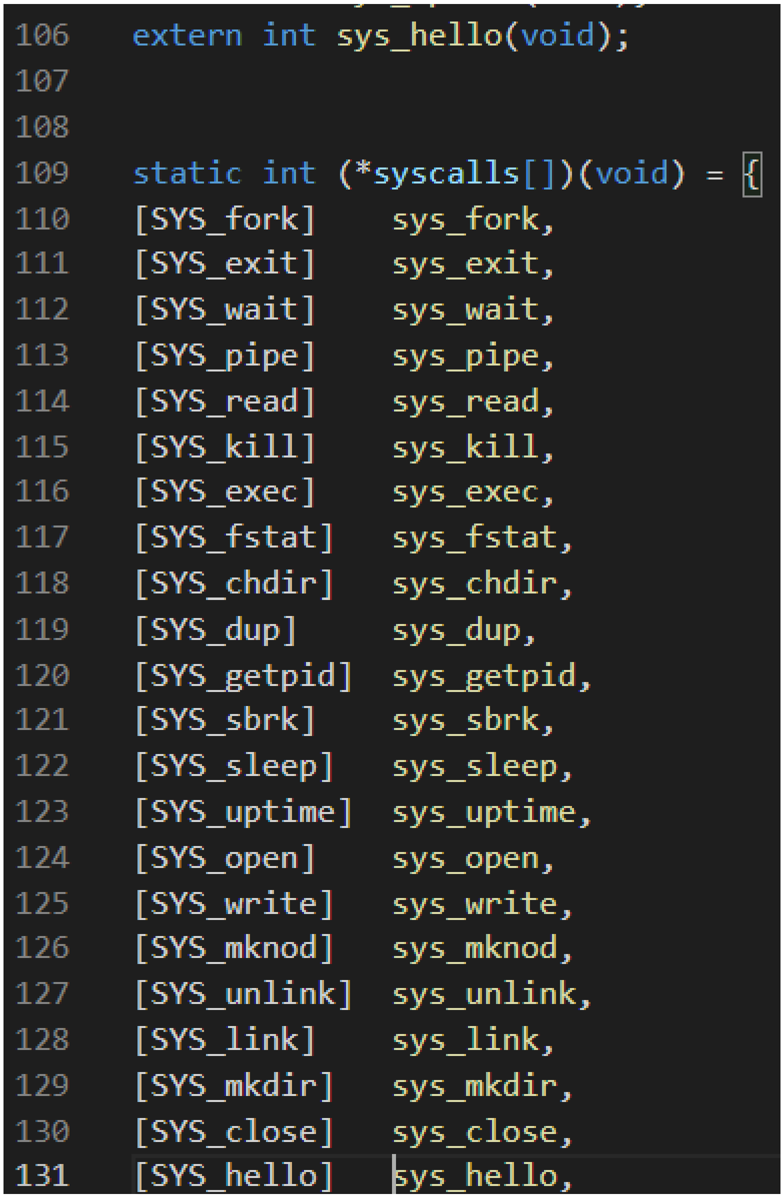
After installing xv6, a few commands were tested to ensure that the installation was successful. The command “cat README” opened the readme document as it should. Then the commands of “mkdir foo” was entered, which created a new directory called “foo”. Then the command to list the files of a directory was issued to verify that “foo” had been created. Once the creation of the directory “foo” was verified, the newly created directory was removed with “rm foo”.



```
$ mkdir foo
$ ls
.              1 1 512
..             1 1 512
README         2 2 2327
cat            2 3 13688
echo           2 4 12692
forktest       2 5 8128
grep           2 6 15564
init           2 7 13280
kill           2 8 12748
ln             2 9 12648
ls             2 10 14832
mkdir          2 11 12828
rm             2 12 12808
sh             2 13 23296
stressfs       2 14 13476
usertests      2 15 56408
wc             2 16 14224
zombie         2 17 12472
hello          2 18 12420
console        3 19 0
foo            1 20 32
$
```

Edit syscall.c

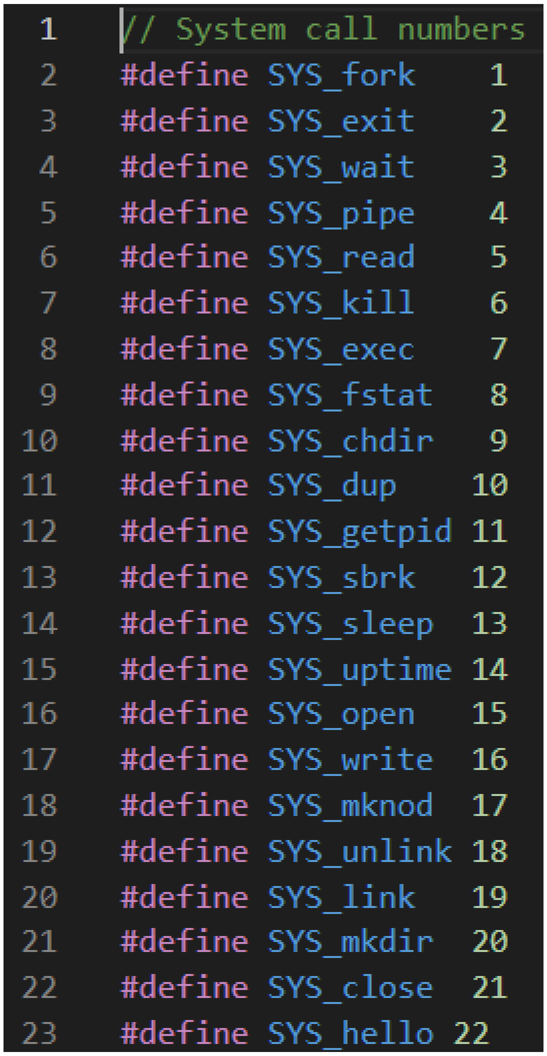
The syscall.c file was edited so that a new system call called “sys_hello” can be called from the terminal.



```
extern int sys_hello(void);


static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_hello]   sys_hello,
```

Edit syscall.h

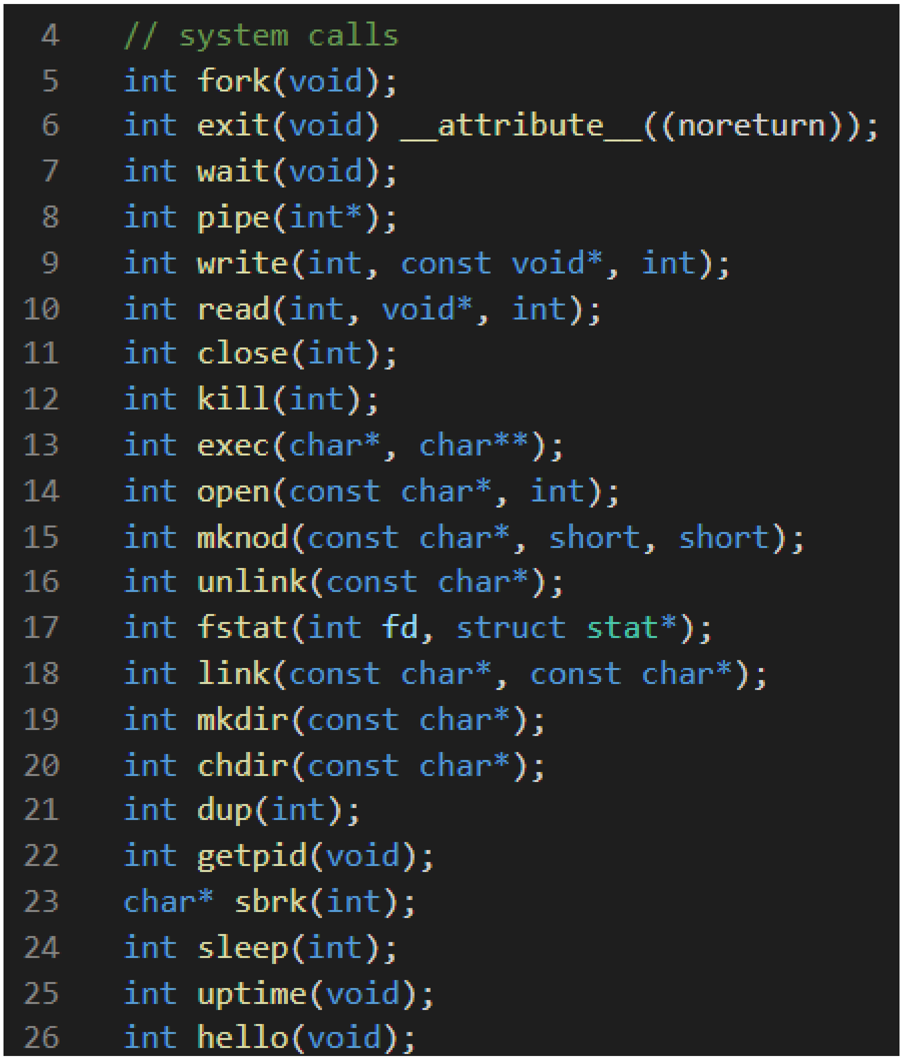
The prototypes for syscall are modified to include the new system call with its own unique number.



```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_hello 22
```

Edit user.h

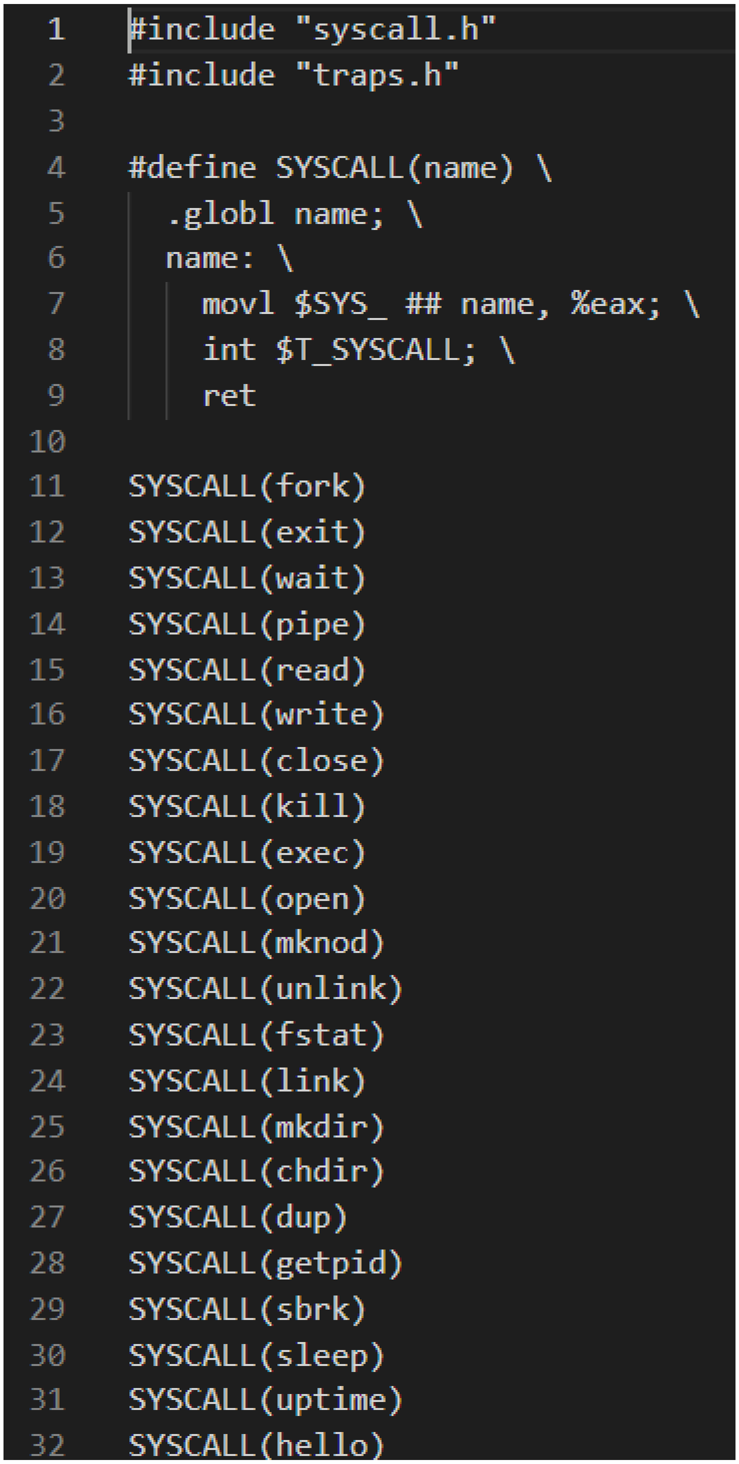
This declares the “hello” function for the user.



```
// system calls
int fork(void);
int exit(void) __attribute__((noreturn));
int wait(void);
int pipe(int*);
int write(int, const void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(const char*, int);
int mknod(const char*, short, short);
int unlink(const char*);
int fstat(int fd, struct stat*);
int link(const char*, const char*);
int mkdir(const char*);
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int hello(void);
```

Edit usys.S

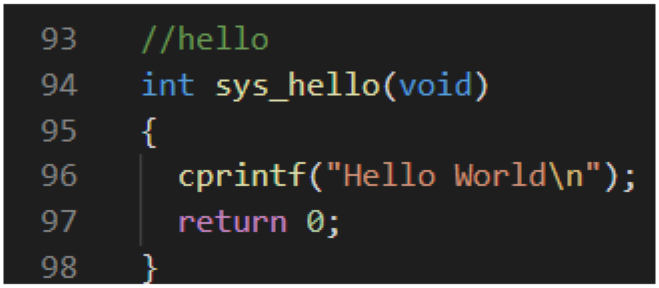
This code will create “hello” as a system call, so that the OS will know how to generate an interrupt and label.



```
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(hello)
```

Edit sysproc.c

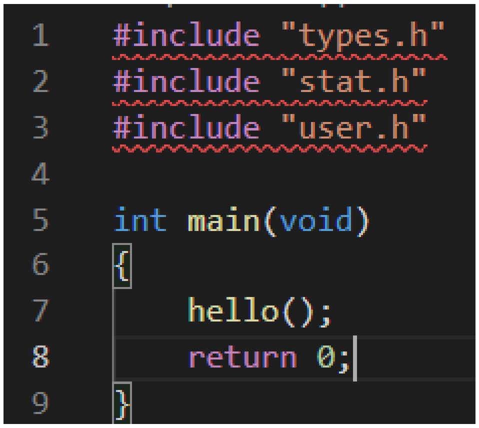
This will be the process from the system call when “hello” is executed.



```
//hello
int sys_hello(void)
{
  cprintf("Hello World\n");
  return 0;
}
```

Make hello.c

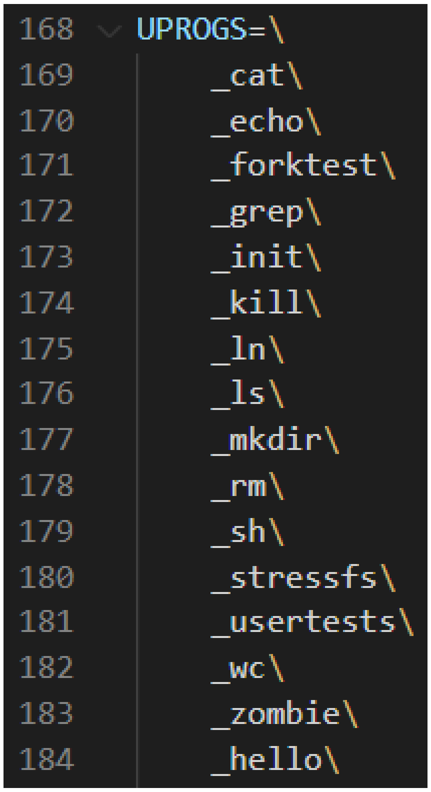
This file calls the “hello” function that is declared in “user.h”.



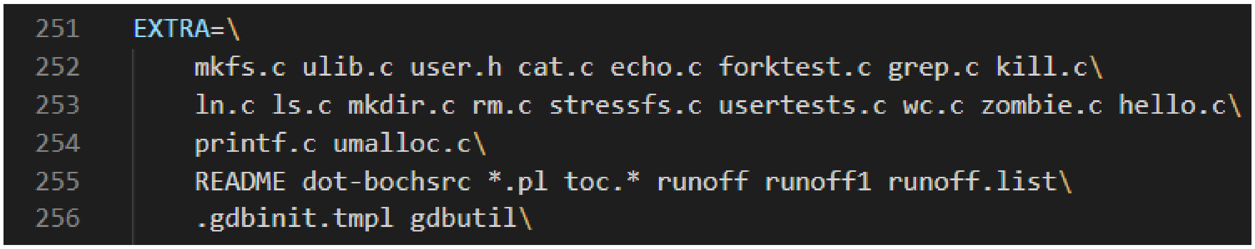
```c
#include "types.h"
#include "stat.h"
#include "user.h"

int main(void)
{
    hello();
    return 0;
}
```

Edit Makefile

Finally, the make file is edited to compile the newly added code in all the files. Plus the newly created “hello.c” must be added to the compilation list to create a new “hello” object file.



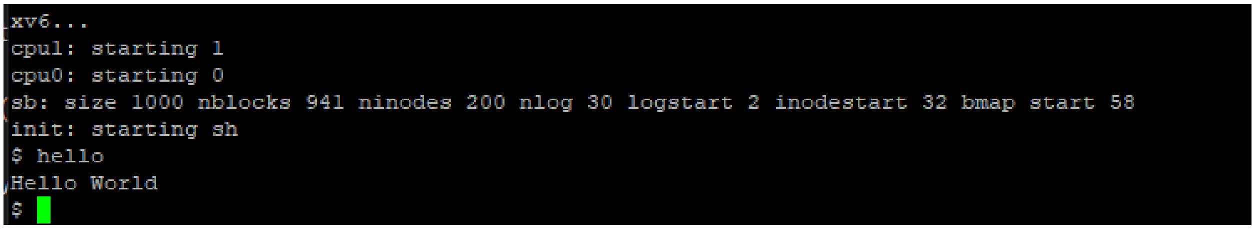
```
UPROGS=\
	_cat\
	_echo\
	_forktest\
	_grep\
	_init\
	_kill\
	_ln\
	_ls\
	_mkdir\
	_rm\
	_sh\
	_stressfs\
	_usertests\
	_wc\
	_zombie\
	_hello\
```



```
EXTRA=\
	mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
	ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c hello.c\
	printf.c umalloc.c\
	README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
	.gdbinit.tmpl gdbutil\
```

Test hello command

This command in xv6 demonstrates that the newly created system call does work in the operating system. When the user types hello, the sys_hello command is given that will print back “Hello World”.



```
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ hello
Hello World
$ 
```

Conclusion

I learned how to add a system call to xv6. I still do not understand fully what all the files do that the code was added to, but I now know the areas that need to be changed to create a new system call. I also know where to look to modify some existing system calls. Overall, this lab allowed me to see how xv6 works. I think it went into more depth than I was expecting for an introduction to xv6 and I would have liked to learn more about the purpose of the files before modifying them. The lab was fun, and being able to see a real effect on an operating system within an hour class was enjoyable.