Adding System Calls to OS/161

CSE 4001 Operating Systems Concepts

E. Ribeiro

January 26, 2022

Outline

Review: System-call trapping mechanism

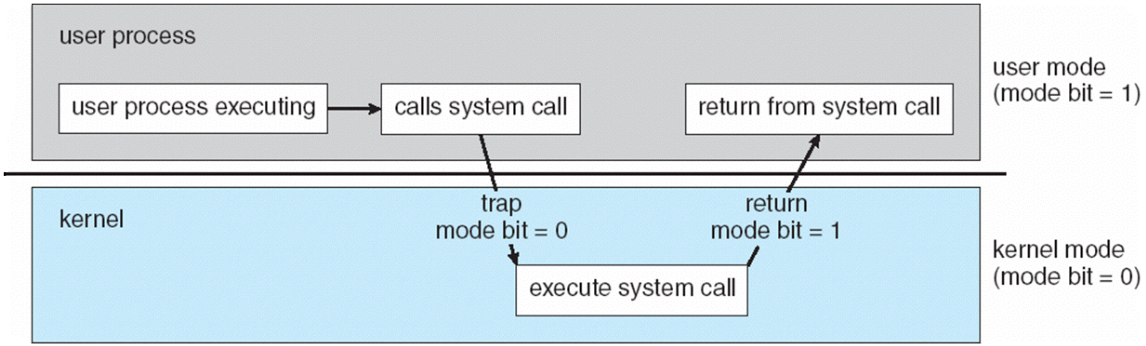
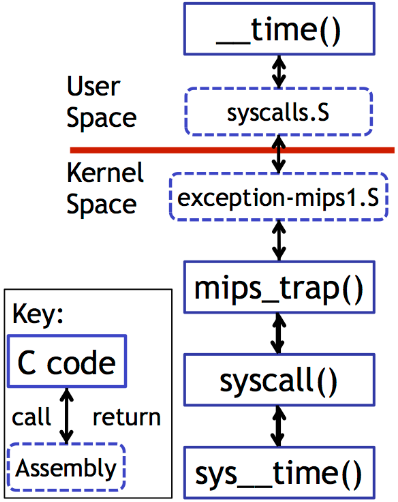


Figure adapted from Silberschatz, Galvin, and Gagne, 2009.

Review: System-call trapping mechanism in OS/161



Kernel-level steps

1. Add the prototype of the system-call function to the header file: `kern/include/syscall.h`
2. The kernel-level implementation (e.g., `newsyscall.c`) goes into `kern/syscall/`
3. Add a new ID number for the system call. The new entry goes in the file `kern/include/kern/syscall.h`
4. Add a new branch in the switch-case statement in: `kern/arch/mips/syscall/syscall.c`
5. Add file entry definition for `syscall/newsyscall.c` in `kern/conf/conf.kern`

User-level steps

1. Add the user-level prototype of the system call to: `user/include/unistd.h`
2. Add the user-level test function. For this, create a new subdirectory directory `user/testbin/testnewsyscall/` and inside it add the test function (e.g., `testnewsyscall.c`).
3. Create a Makefile inside this subdirectory for building the test function. You can use one of the subdirectories as a template.
4. Add an entry to the new function to the top-level Makefile in `user/testbin`

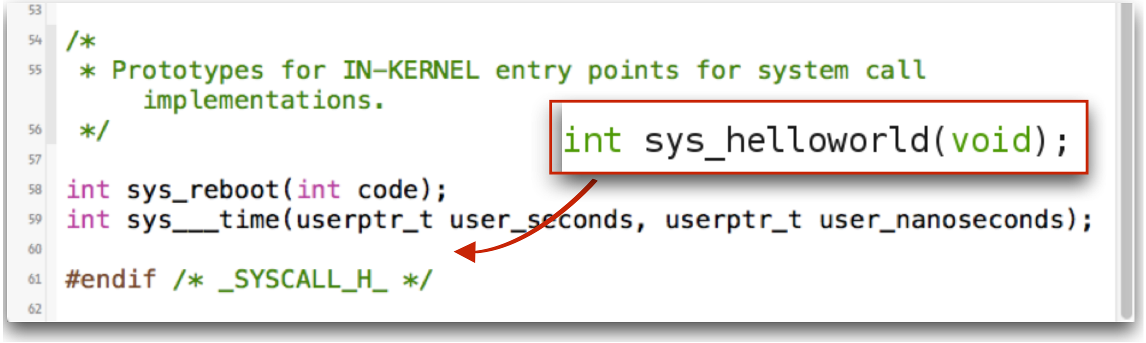
Testing the new system call

1. Re-build the kernel
2. Start the new kernel (i.e., run sys161 kernel in the root directory)
3. At the OS161 prompt, use the p option (from OS161 menu) to run the test program, i.e., `p testbin/testnewsyscall`

Kernel-level steps

1 Prototype of the system call

1. Add the prototype of the system call to the header file: `kern/include/syscall.h`
2. At the end of the file, you will find prototypes for `sys_reboot()` and `sys___time()`.

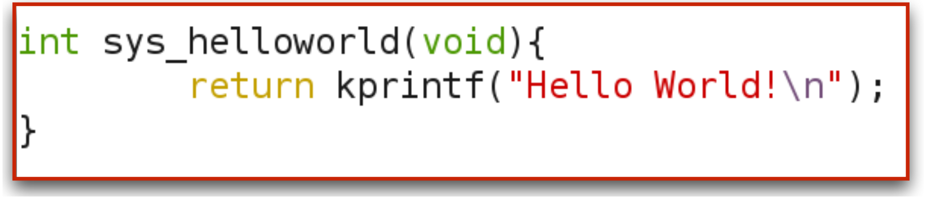


```
/*
 * Prototypes for IN-KERNEL entry points for system call
     implementations.
 */

int sys_reboot(int code);
int sys___time(userptr_t user_seconds, userptr_t user_nanoseconds);

#endif /* _SYSCALL_H_ */
```

```
int sys_helloworld(void);
```

2 Kernel-level implementation

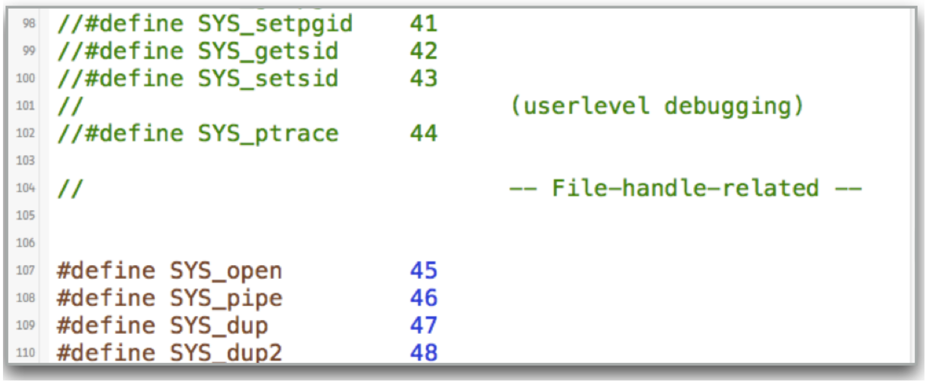
1. The kernel-level implementation goes into `kern/syscall`. This directory contains an example of a system call, i.e., `time_syscalls.c`.
2. Here, create a program called `simple_syscall.c`, and implement your system call in it.



```
int sys_helloworld(void){
        return kprintf("Hello World!\n");
}
```

3 Create the ID number for the new system call

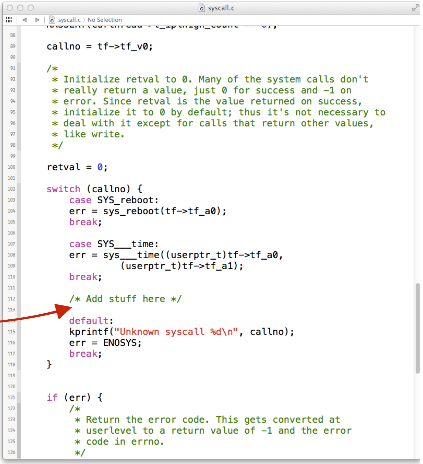
1. The OS needs to know the ID number of the system call
2. Add a new entry to the file `kern/include/kern/syscall.h`



```
//#define SYS_setpgid    41
//#define SYS_getsid     42
//#define SYS_setsid     43
//                              (userlevel debugging)
//#define SYS_ptrace     44

//                              -- File-handle-related --


#define SYS_open         45
#define SYS_pipe         46
#define SYS_dup          47
#define SYS_dup2         48
```

4 Add a new branch in the switch-case statement in: `kern/arch/mips/syscall/syscall.c`

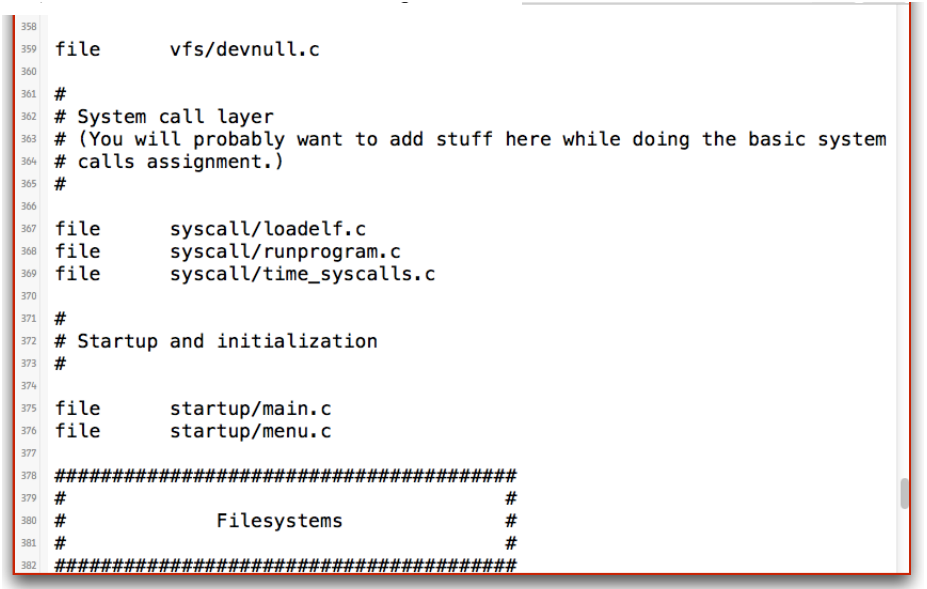
```
case SYS_helloworld:
    err = sys_helloworld();
    break;
```



```
    callno = tf->tf_v0;

    /*
     * Initialize retval to 0. Many of the system calls don't
     * really return a value, just 0 for success and -1 on
     * error. Since retval is the value returned on success,
     * initialize it to 0 by default; thus it's not necessary to
     * deal with it except for calls that return other values,
     * like write.
     */

    retval = 0;

    switch (callno) {
        case SYS_reboot:
        err = sys_reboot(tf->tf_a0);
        break;

        case SYS___time:
        err = sys___time((userptr_t)tf->tf_a0,
                (userptr_t)tf->tf_a1);
        break;

        /* Add stuff here */

        default:
        kprintf("Unknown syscall %d\n", callno);
        err = ENOSYS;
        break;
    }


    if (err) {
        /*
         * Return the error code. This gets converted at
         * userlevel to a return value of -1 and the error
         * code in errno.
         */
```

Note how user-level input parameters are passed to kernel-level functions via the trapframe.

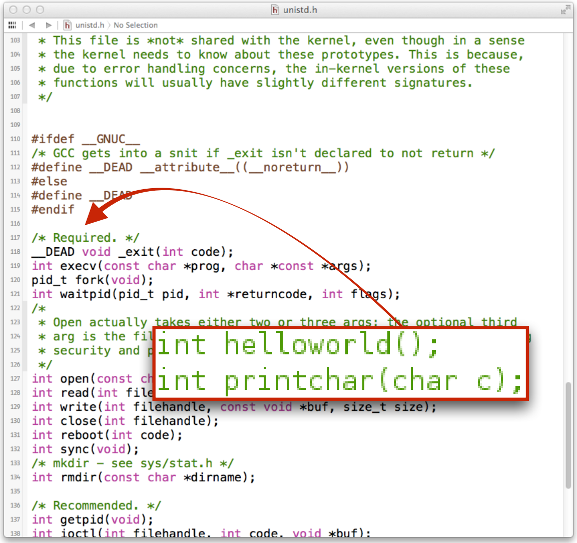
5 Add file-entry definition to config.kern



```

file        vfs/devnull.c

#
# System call layer
# (You will probably want to add stuff here while doing the basic system
# calls assignment.)
#

file        syscall/loadelf.c
file        syscall/runprogram.c
file        syscall/time_syscalls.c

#
# Startup and initialization
#

file        startup/main.c
file        startup/menu.c

########################################
#                                      #
#             Filesystems              #
#                                      #
########################################
```

User-level steps

1. Add the user-level prototype of the system call to: `userland/include/unistd.h`

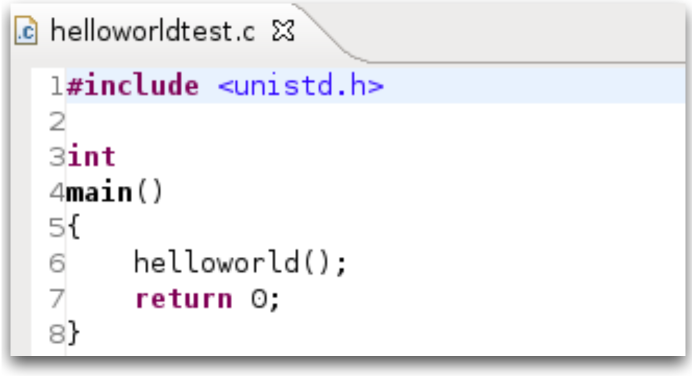


```
 * This file is *not* shared with the kernel, even though in a sense
 * the kernel needs to know about these prototypes. This is because,
 * due to error handling concerns, the in-kernel versions of these
 * functions will usually have slightly different signatures.
 */


#ifdef __GNUC__
/* GCC gets into a snit if _exit isn't declared to not return */
#define __DEAD __attribute__((__noreturn__))
#else
#define __DEAD
#endif

/* Required. */
__DEAD void _exit(int code);
int execv(const char *prog, char *const *args);
pid_t fork(void);
int waitpid(pid_t pid, int *returncode, int flags);
/*
 * Open actually takes either two or three args: the optional third
 * arg is the fil
 * security and p
 */
int open(const ch
int read(int file
int write(int filehandle, const void *buf, size_t size);
int close(int filehandle);
int reboot(int code);
int sync(void);
/* mkdir - see sys/stat.h */
int rmdir(const char *dirname);

/* Recommended. */
int getpid(void);
int ioctl(int filehandle, int code, void *buf);
```

```
int helloworld();
int printchar(char c);
```

2. Add the user-level test function.

For this, create a new subdirectory directory `user/testbin/testnewsyscall/` and inside it add the test function (e.g., `testnewsyscall.c`).

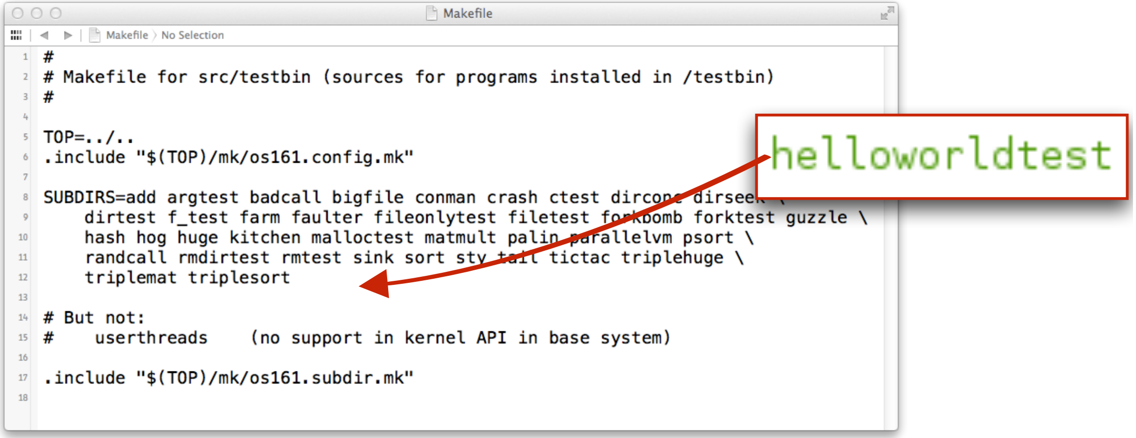


```c
#include <unistd.h>

int
main()
{
    helloworld();
    return 0;
}
```

```make
# Makefile for helloworldtest

TOP=../../..
.include "$(TOP)/mk/os161.config.mk"

PROG=helloworldtest
SRCS=helloworldtest.c
BINDIR=/testbin

.include "$(TOP)/mk/os161.prog.mk"
```

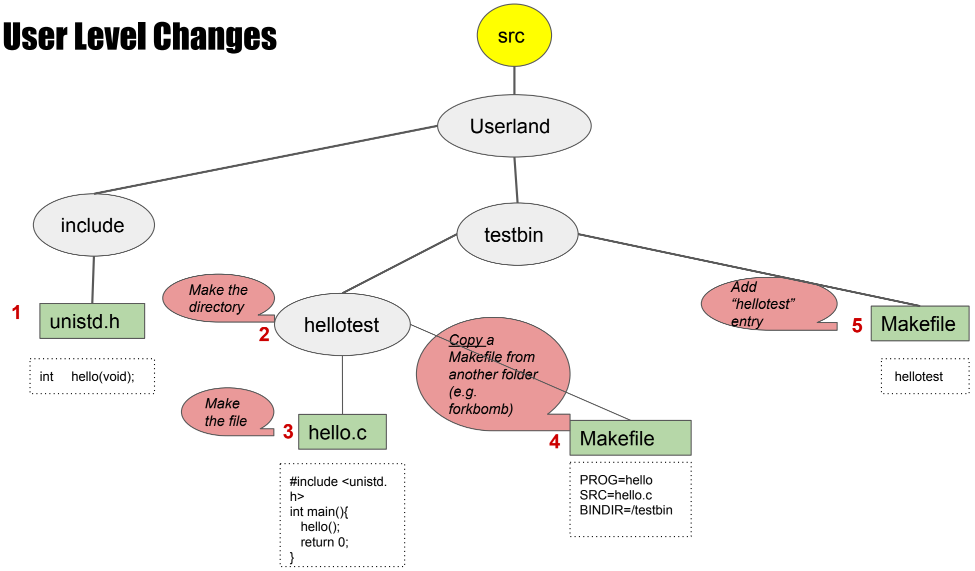
3. Modify the top-level makefile.

Add an entry to the new function to the top-level Makefile in `user/testbin/`



```
#
# Makefile for src/testbin (sources for programs installed in /testbin)
#

TOP=../..
.include "$(TOP)/mk/os161.config.mk"

SUBDIRS=add argtest badcall bigfile conman crash ctest dircone dirseek \
    dirtest f_test farm faulter fileonlytest filetest forkbomb forktest guzzle \
    hash hog huge kitchen malloctest matmult palin parallelvm psort \
    randcall rmdirtest rmtest sink sort sty tail tictac triplehuge \
    triplemat triplesort

# But not:
#    userthreads    (no support in kernel API in base system)

.include "$(TOP)/mk/os161.subdir.mk"
```

Directory tree showing main changes that need to be made



```
int     hello(void);
```

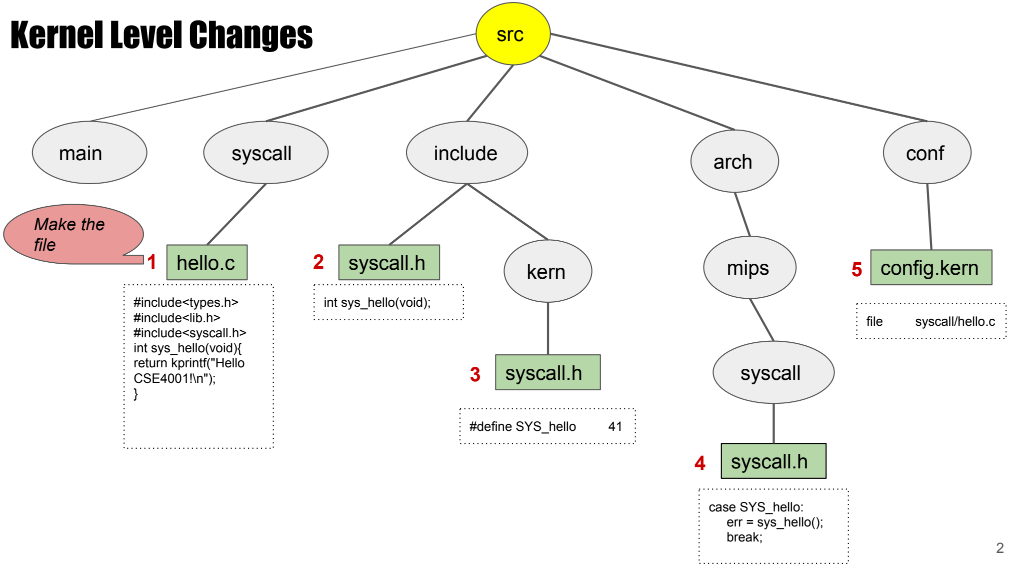
```
#include <unistd.
h>
int main(){
   hello();
   return 0;
}
```

```
PROG=hello
SRC=hello.c
BINDIR=/testbin
```

```
hellotest
```

Directory tree showing main changes that need to be made

Kernel Level Changes



```
#include<types.h>
#include<lib.h>
#include<syscall.h>
int sys_hello(void){
return kprintf("Hello
CSE4001!\n");
}
```

```
int sys_hello(void);
```

```
#define SYS_hello        41
```

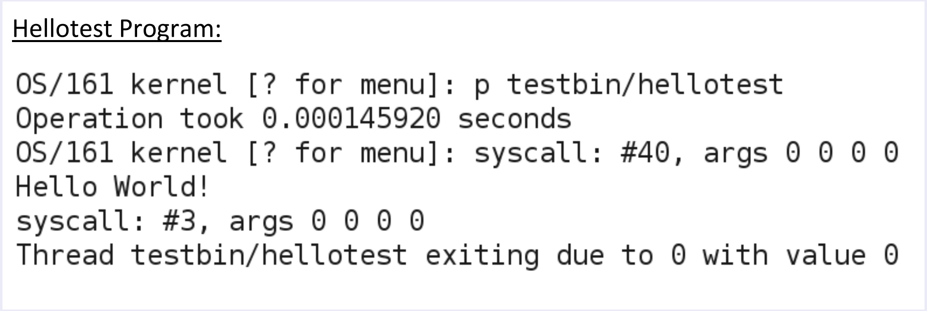
```
case SYS_hello:
     err = sys_hello();
     break;
```

```
file        syscall/hello.c
```

Testing the system call

Testing the system call

1. Inside the root folder, run the command `sys161 kernel`.
2. In the os161 terminal, run the command `p testbin/[name]` where your [name] is the name of your program.



<u>Hellotest Program:</u>

```
OS/161 kernel [? for menu]: p testbin/hellotest
Operation took 0.000145920 seconds
OS/161 kernel [? for menu]: syscall: #40, args 0 0 0 0
Hello World!
syscall: #3, args 0 0 0 0
Thread testbin/hellotest exiting due to 0 with value 0
```