System Calls

What are they?

- Standard interface to allow the kernel to safely handle use requests. Example requests:
  - Read from hardware
  - Spawn a new process
  - Get current time
  - Create shared memory

- Message passing technique between:
  - OS kernel (server)
  - User (client)

Executing System Calls

- User program issues call
- Core kernel looks up the call in the syscall table
- *Kernel module*handles syscall action
- Module returns result of system call
- Core kernel forwards results to the user

What if the module is not loaded?

- User program issues call
- Core kernel looks up the call in the syscall table
- Kernel module isn’t loaded to handle action
- …

- Where does call go?

System Call Wrappers

- Have a system call wrapper to handle this scenario.
- Wrapper calls the system call handler function if the module is loaded
- Else, it returns an error

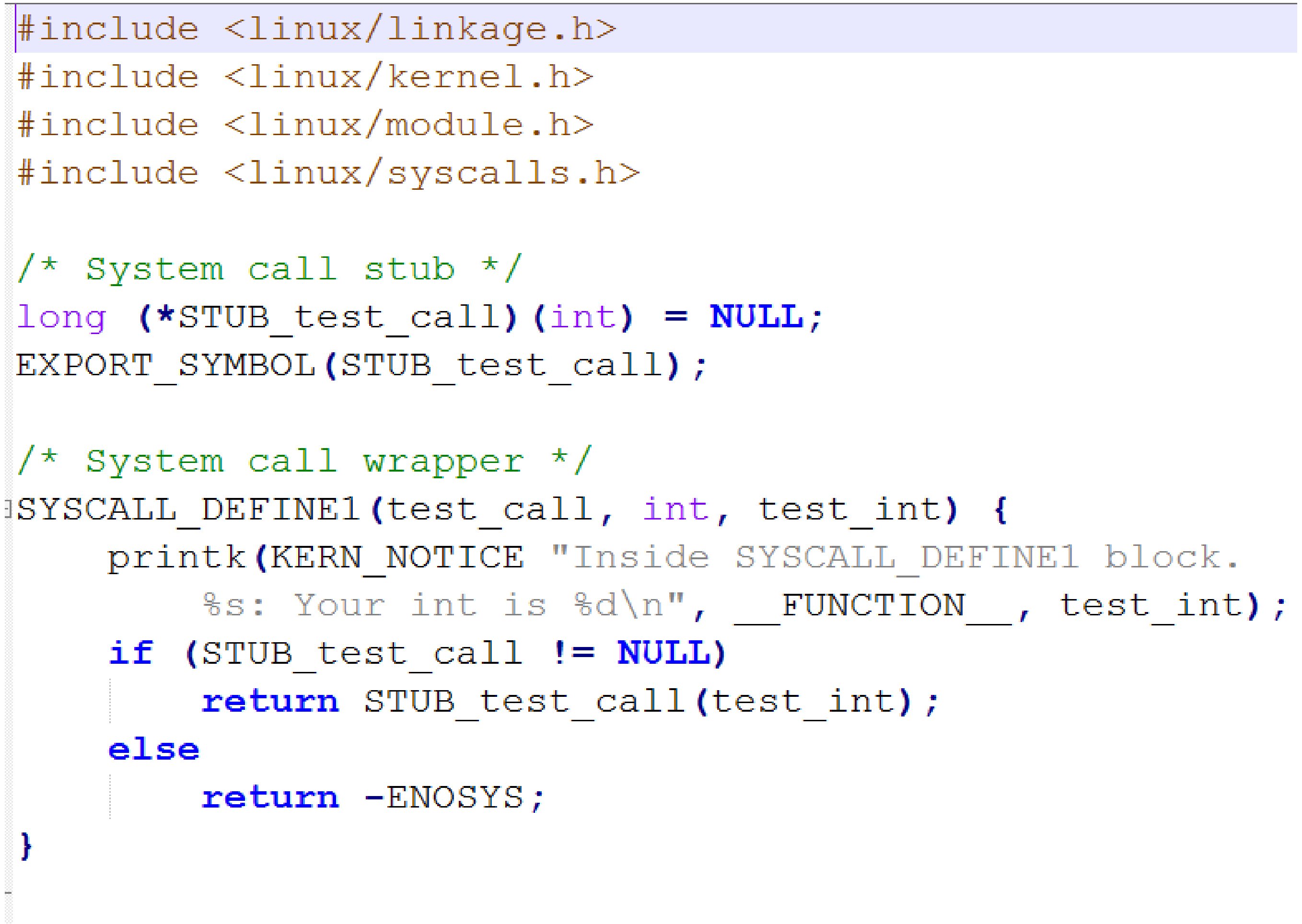
- Uses a function pointer to point the system call handler function

- Add a system call wrapper for each system call you add.

Adding System Calls

- Files to add:
  - /usr/src/test_kernel/SystemCalls/test_call.c
  - /usr/src/test_kernel/SystemCalls/Makefile
  - /usr/src/test_kernel/SystemCalls/syscallModule.c
- Files to modify:
  - /usr/src/test_kernel/*arch/x86/entry/syscalls/syscall_64.tbl*
  - /usr/src/test_kernel/*include/linux/syscalls.h*
  - /usr/src/test_kernel/Makefile

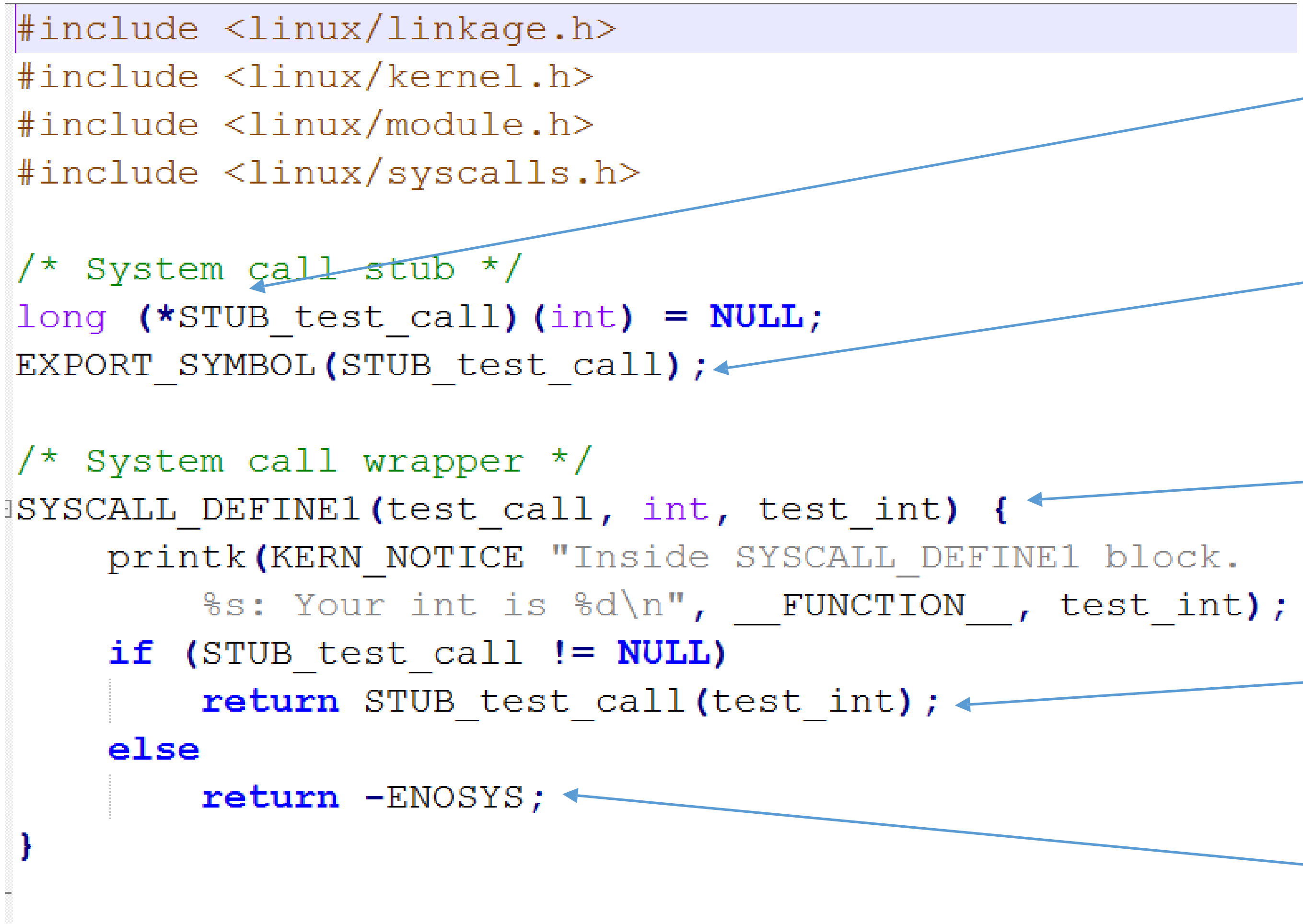
SystemCalls/test_call.c



```c
#include <linux/linkage.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/syscalls.h>

/* System call stub */
long (*STUB_test_call)(int) = NULL;
EXPORT_SYMBOL(STUB_test_call);

/* System call wrapper */
SYSCALL_DEFINE1(test_call, int, test_int) {
    printk(KERN_NOTICE "Inside SYSCALL_DEFINE1 block.
        %s: Your int is %d\n", __FUNCTION__, test_int);
    if (STUB_test_call != NULL)
        return STUB_test_call(test_int);
    else
        return -ENOSYS;
}
```

- Creates syscall pointer.
- Exports the pointer so that the system call module can access it.
- Define syscall wrapper.

SystemCalls/test_call.c



```
#include <linux/linkage.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/syscalls.h>

/* System call stub */
long (*STUB_test_call)(int) = NULL;
EXPORT_SYMBOL(STUB_test_call);

/* System call wrapper */
SYSCALL_DEFINE1(test_call, int, test_int) {
    printk(KERN_NOTICE "Inside SYSCALL_DEFINE1 block.
        %s: Your int is %d\n", __FUNCTION__, test_int);
    if (STUB_test_call != NULL)
        return STUB_test_call(test_int);
    else
        return -ENOSYS;
}
```

syscall pointer (function pointer)

Exports syscall pointer so that the handler module can access it

Wrapper function

Execute if defined

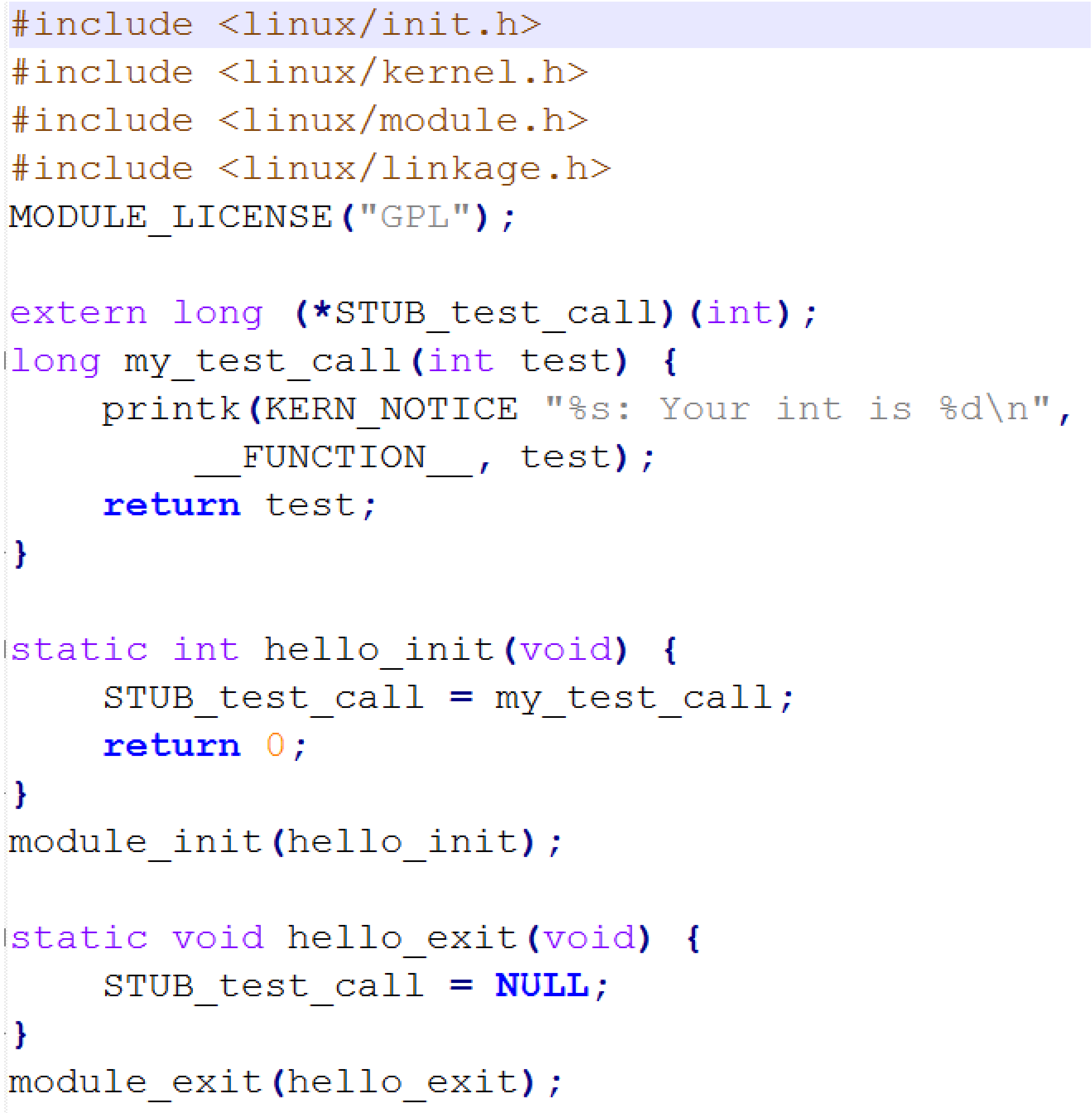
Return error if not defined.

SYSCALL_DEFINEn

- SYSCALL_DEFINEn is a macro the generates the proper system call definition with the appropriate type of arguments.
- In SYSCALL_DEFINE1, *1* means the system call will take 1 argument.
- *SYSCALL_DEFINE1(test_call, int, test_int)* creates a system call named *sys_test_call*, which takes one argument *test_int*which is of type *int.* (Notice the comma between*int*and*test_int).*

- SYSCALL_DEFINE0 to SYSCALL_DEFINE6 are defined, so you can pass from zero to maximum six arguments to a system call.

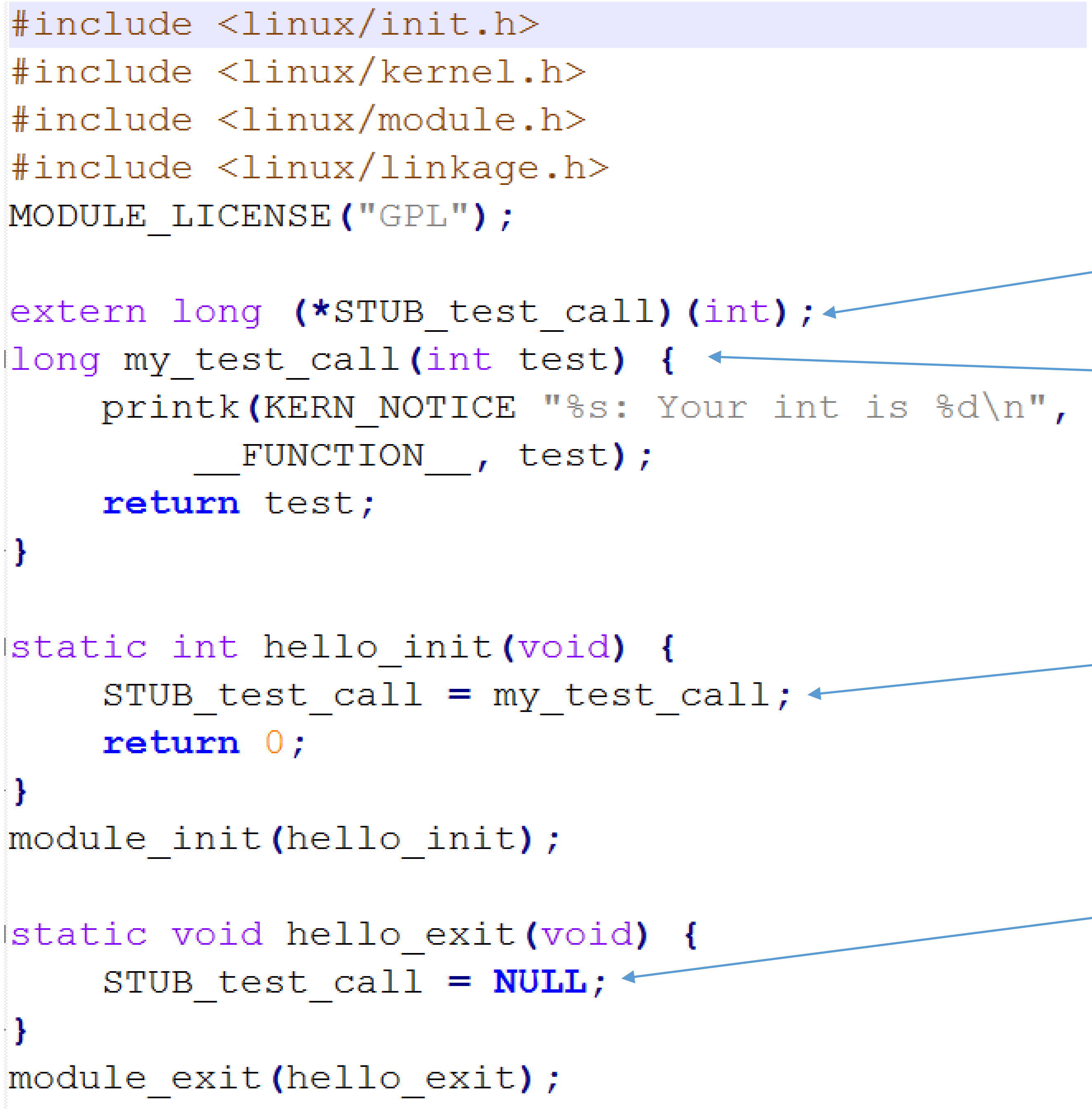
SystemCalls/syscallModule.c



```c
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/linkage.h>
MODULE_LICENSE("GPL");

extern long (*STUB_test_call)(int);
long my_test_call(int test) {
    printk(KERN_NOTICE "%s: Your int is %d\n",
        __FUNCTION__, test);
    return test;
}

static int hello_init(void) {
    STUB_test_call = my_test_call;
    return 0;
}
module_init(hello_init);

static void hello_exit(void) {
    STUB_test_call = NULL;
}
module_exit(hello_exit);
```

- Holds module code.
- Registers syscall pointer to the proper syscall handler.
- Implements syscall behavior.

SystemCalls/syscallModule.c



```c
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/linkage.h>
MODULE_LICENSE("GPL");

extern long (*STUB_test_call)(int);
long my_test_call(int test) {
    printk(KERN_NOTICE "%s: Your int is %d\n",
        __FUNCTION__, test);
    return test;
}

static int hello_init(void) {
    STUB_test_call = my_test_call;
    return 0;
}
module_init(hello_init);

static void hello_exit(void) {
    STUB_test_call = NULL;
}
module_exit(hello_exit);
```

Get access to the syscall pointer

Actual system call handler.

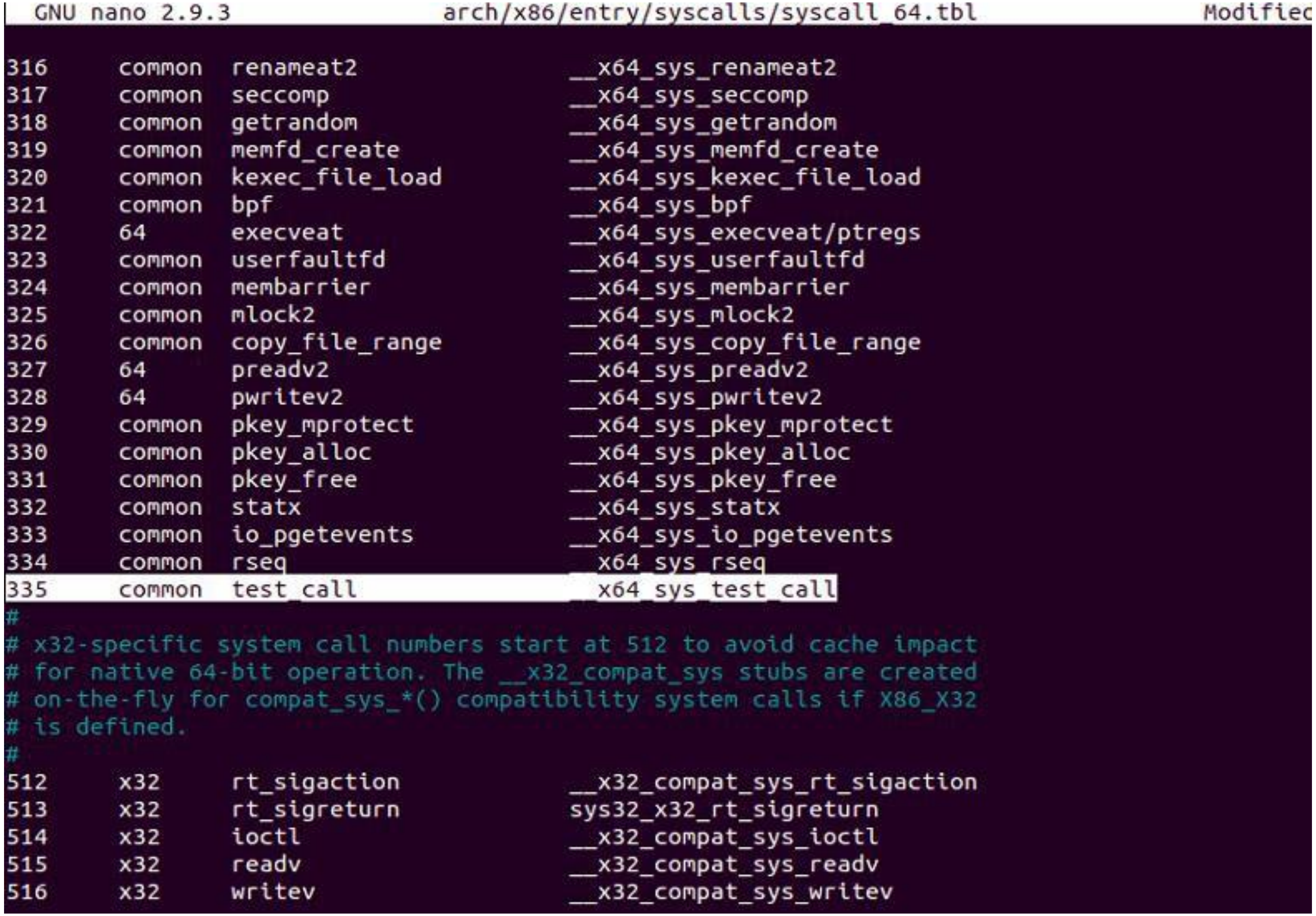
Assigns the syscall pointer to the syscall handler function when the module is loaded.

Empties the syscall pointer when the module is unloaded.

SystemCalls/Makefile

```
obj-y :=test_call.o
obj-m := syscallModule.o

PWD := $(shell pwd)
KDIR := /lib/modules/`uname -r`/build

default:
        $(MAKE) –C $(KDIR) SUBDIRS= $(PWD) modules

clean:
        rm –f *.o *.ko *.mod.* Module.* modules.*
```

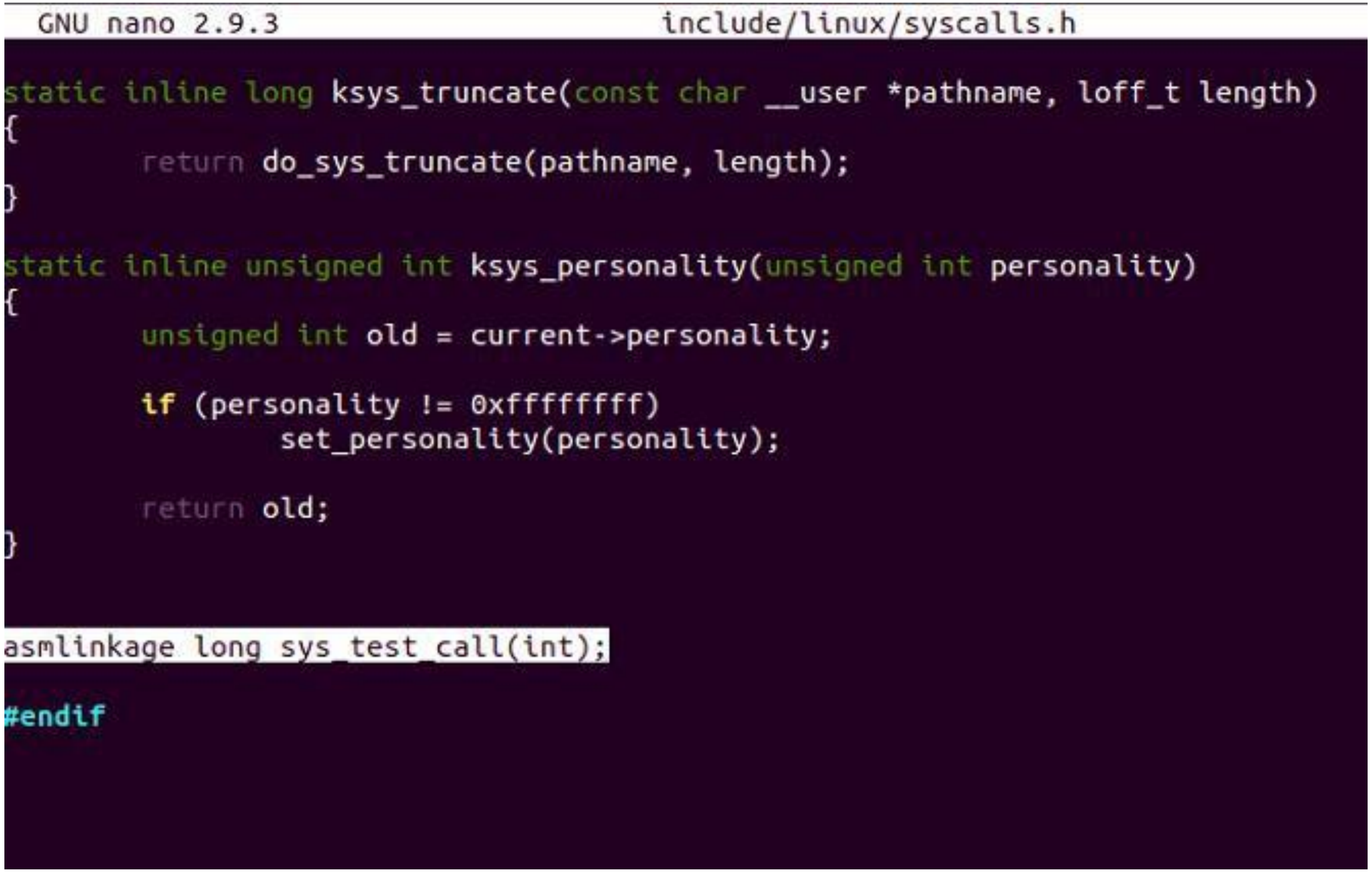
arch/x86/entry/syscalls/syscall_64.tbl



```
  GNU nano 2.9.3              arch/x86/entry/syscalls/syscall_64.tbl              Modified

316     common  renameat2               __x64_sys_renameat2
317     common  seccomp                 __x64_sys_seccomp
318     common  getrandom               __x64_sys_getrandom
319     common  memfd_create            __x64_sys_memfd_create
320     common  kexec_file_load         __x64_sys_kexec_file_load
321     common  bpf                     __x64_sys_bpf
322     64      execveat                __x64_sys_execveat/ptregs
323     common  userfaultfd             __x64_sys_userfaultfd
324     common  membarrier              __x64_sys_membarrier
325     common  mlock2                  __x64_sys_mlock2
326     common  copy_file_range         __x64_sys_copy_file_range
327     64      preadv2                 __x64_sys_preadv2
328     64      pwritev2                __x64_sys_pwritev2
329     common  pkey_mprotect           __x64_sys_pkey_mprotect
330     common  pkey_alloc              __x64_sys_pkey_alloc
331     common  pkey_free               __x64_sys_pkey_free
332     common  statx                   __x64_sys_statx
333     common  io_pgetevents           __x64_sys_io_pgetevents
334     common  rseq                    __x64_sys_rseq
335     common  test_call               __x64_sys_test_call
#
# x32-specific system call numbers start at 512 to avoid cache impact
# for native 64-bit operation. The __x32_compat_sys stubs are created
# on-the-fly for compat_sys_*() compatibility system calls if X86_X32
# is defined.
#
512     x32     rt_sigaction            __x32_compat_sys_rt_sigaction
513     x32     rt_sigreturn            sys32_x32_rt_sigreturn
514     x32     ioctl                   __x32_compat_sys_ioctl
515     x32     readv                   __x32_compat_sys_readv
516     x32     writev                  __x32_compat_sys_writev
```

- Here, we add our system call into the table.
- Up to 334 was already there, so we take 335.
- Add the line:
  335<tab>common<tab>test_call <tab>__x64_sys_test_call
- Notice how the name becomes __x64_sys_test_call from our simple test_call

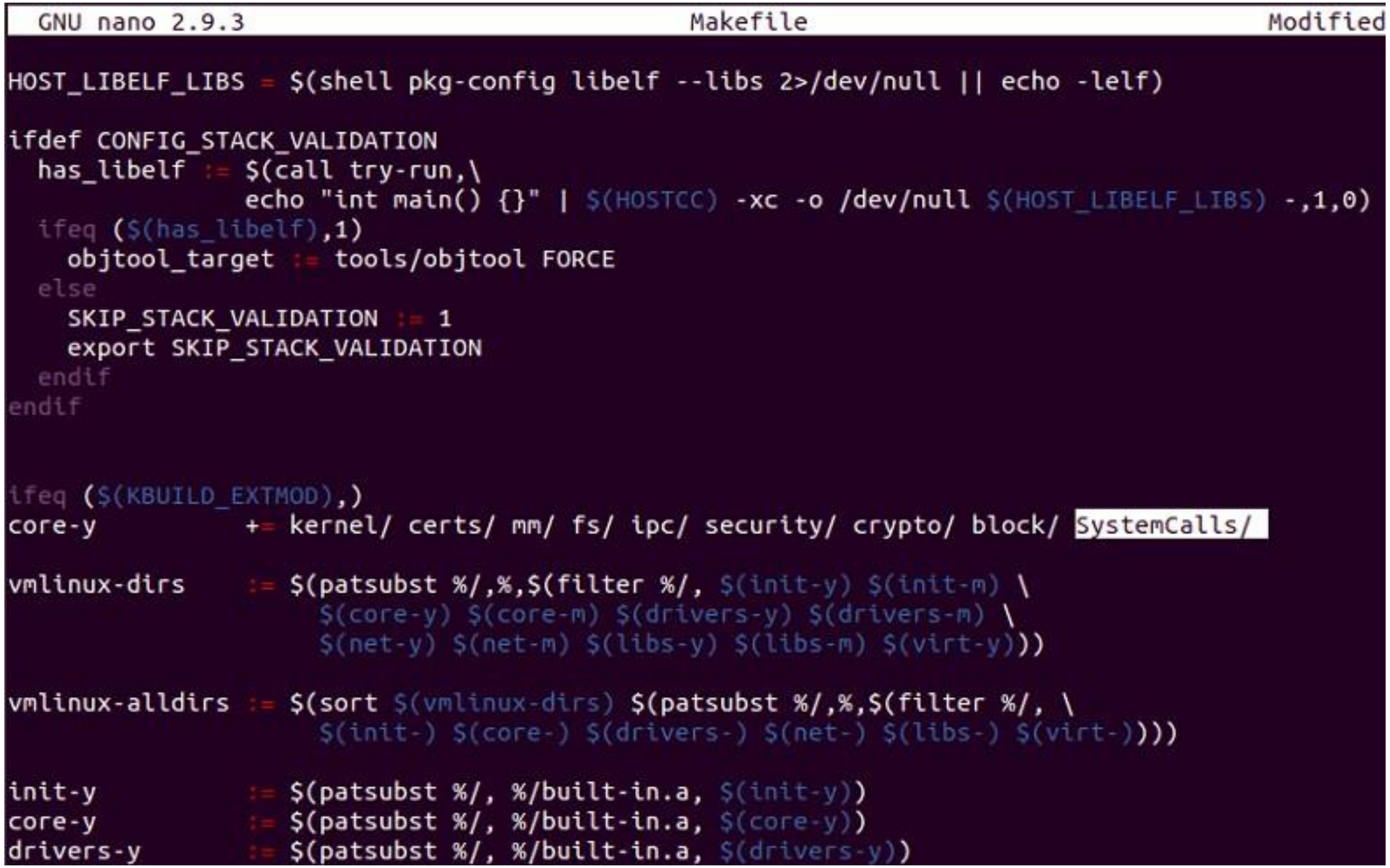
Include/linux/syscalls.h



```
  GNU nano 2.9.3                    include/linux/syscalls.h

static inline long ksys_truncate(const char __user *pathname, loff_t length)
{
        return do_sys_truncate(pathname, length);
}

static inline unsigned int ksys_personality(unsigned int personality)
{
        unsigned int old = current->personality;

        if (personality != 0xffffffff)
                set_personality(personality);

        return old;
}


asmlinkage long sys_test_call(int);

#endif
```

- End of document
- Define the system call prototype
- Notice this time we write sys_test_call, instead of test_call or ______________x64_sys_test_call

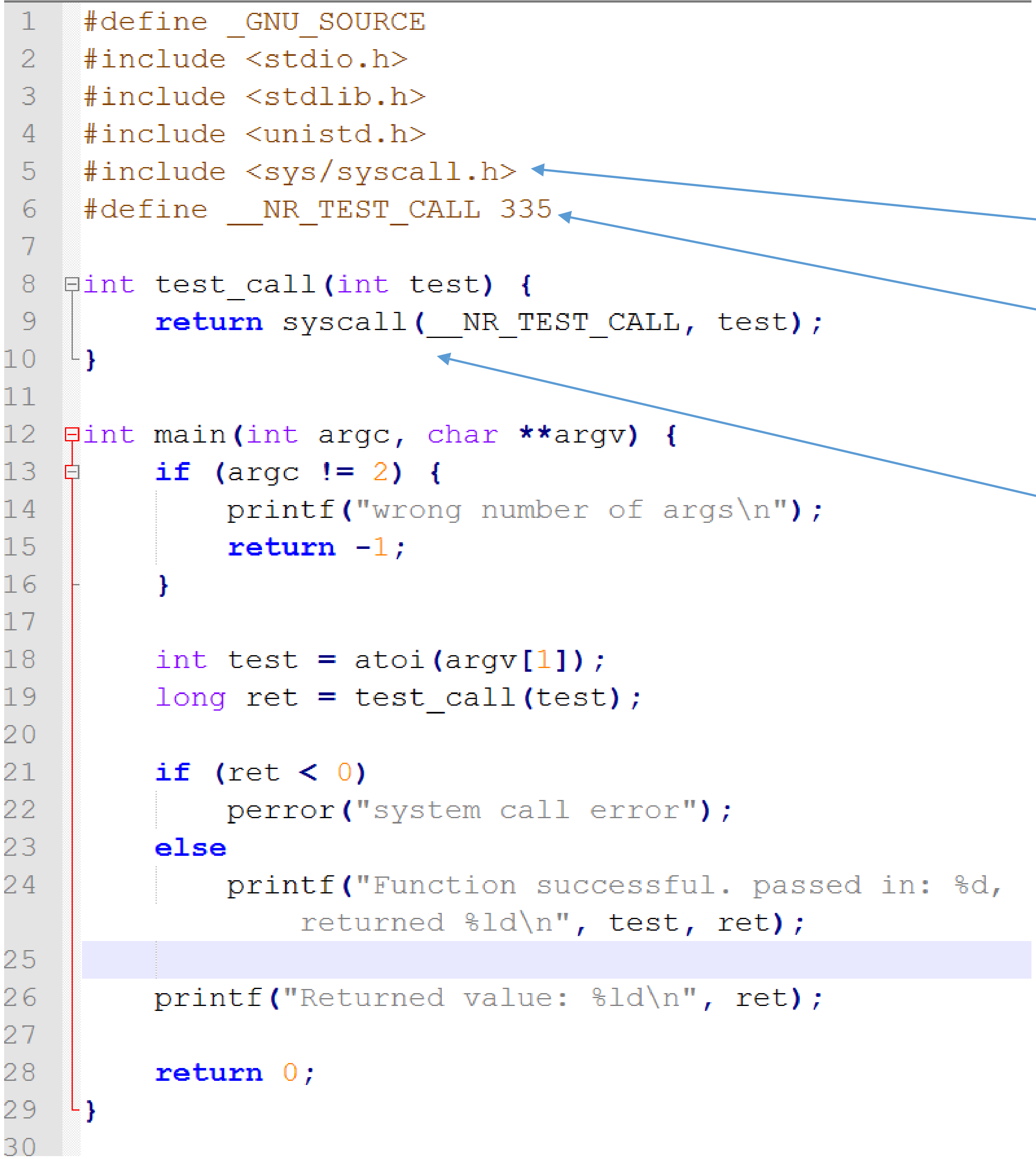
Makefile



```
  GNU nano 2.9.3                              Makefile                                Modified

HOST_LIBELF_LIBS = $(shell pkg-config libelf --libs 2>/dev/null || echo -lelf)

ifdef CONFIG_STACK_VALIDATION
  has_libelf := $(call try-run,\
                echo "int main() {}" | $(HOSTCC) -xc -o /dev/null $(HOST_LIBELF_LIBS) -,1,0)
  ifeq ($(has_libelf),1)
    objtool_target := tools/objtool FORCE
  else
    SKIP_STACK_VALIDATION := 1
    export SKIP_STACK_VALIDATION
  endif
endif


ifeq ($(KBUILD_EXTMOD),)
core-y          += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/ SystemCalls/

vmlinux-dirs    := $(patsubst %/,%,$(filter %/, $(init-y) $(init-m) \
                     $(core-y) $(core-m) $(drivers-y) $(drivers-m) \
                     $(net-y) $(net-m) $(libs-y) $(libs-m) $(virt-y)))

vmlinux-alldirs := $(sort $(vmlinux-dirs) $(patsubst %/,%,$(filter %/, \
                     $(init-) $(core-) $(drivers-) $(net-) $(libs-) $(virt-))))

init-y          := $(patsubst %/, %/built-in.a, $(init-y))
core-y          := $(patsubst %/, %/built-in.a, $(core-y))
drivers-y       := $(patsubst %/, %/built-in.a, $(drivers-y))
```

- Search for the second occurrence of *corey*
- Add the *SystemCalls*directory to the list. These are the directories that have files to be built directly into the kernel.

User space program



```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/syscall.h>
#define __NR_TEST_CALL 335

int test_call(int test) {
    return syscall(__NR_TEST_CALL, test);
}

int main(int argc, char **argv) {
    if (argc != 2) {
        printf("wrong number of args\n");
        return -1;
    }

    int test = atoi(argv[1]);
    long ret = test_call(test);

    if (ret < 0)
        perror("system call error");
    else
        printf("Function successful. passed in: %d,
            returned %ld\n", test, ret);

    printf("Returned value: %ld\n", ret);

    return 0;
}
```

Definition of syscall

Our syscall number

Making a call to the system call.

Notes

- Adding a new system call requires recompiling and reinstalling the whole kernel.
- However, a module can be added at any time without kernel reinstallation.
- That’s why, keep the system call definition function (in our case, test_call.c) really simple and compile only once. Ideally, this function only creates the sys_call pointer and calls it.
- Implement the actual system call handler functions as a module (in our case, syscallModule.c). You can compile it as many times as you want.