Add a syscall to Linux 3.9

@linuxpoetry, github.com/mrrrgn

Follow along at:

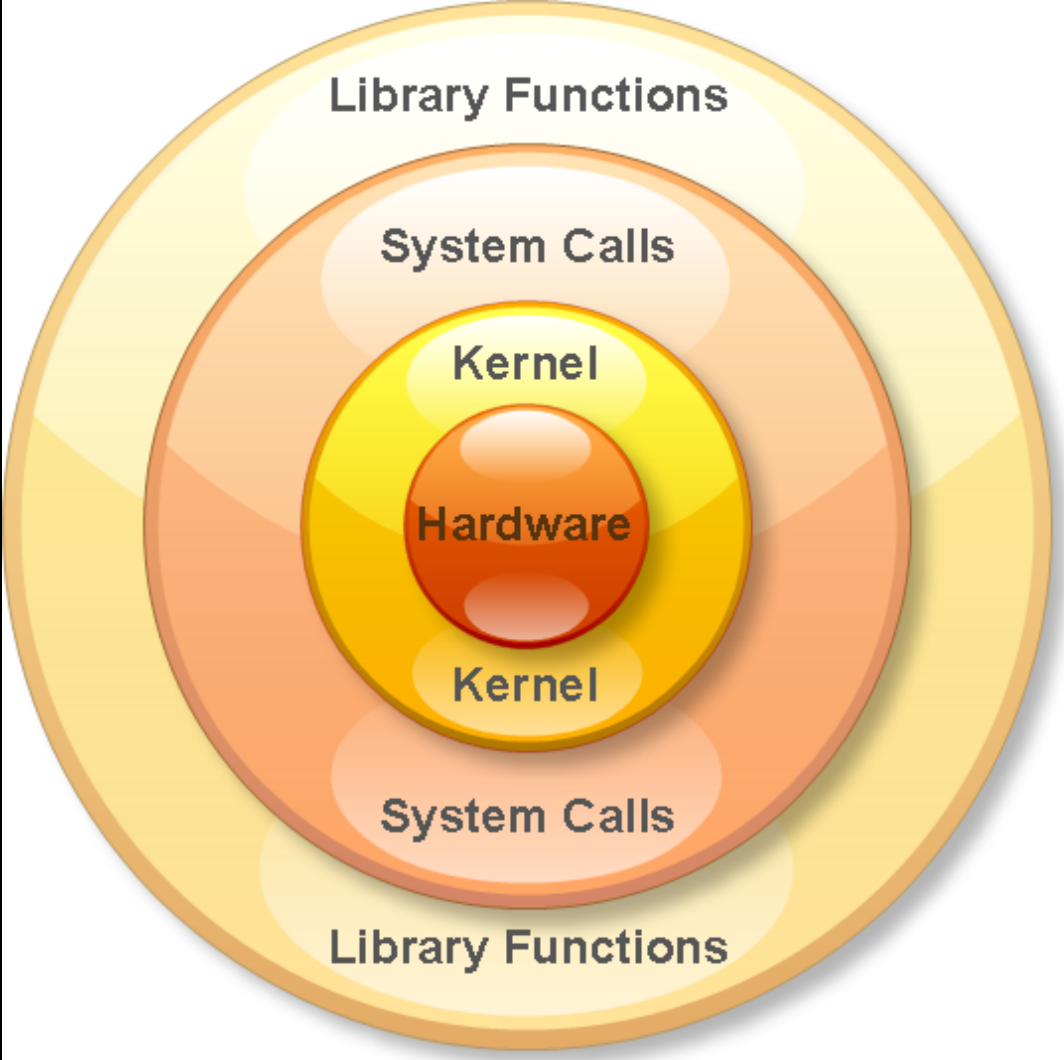
https://github.com/mrrrgn/syscall-3.9

What is a system call?

An interface that exposes kernel logic to user space applications

Used when you need to do something that requires full access to system resources

- input/output
- *process creation*
- *memory allocation*
- *interprocess communication*

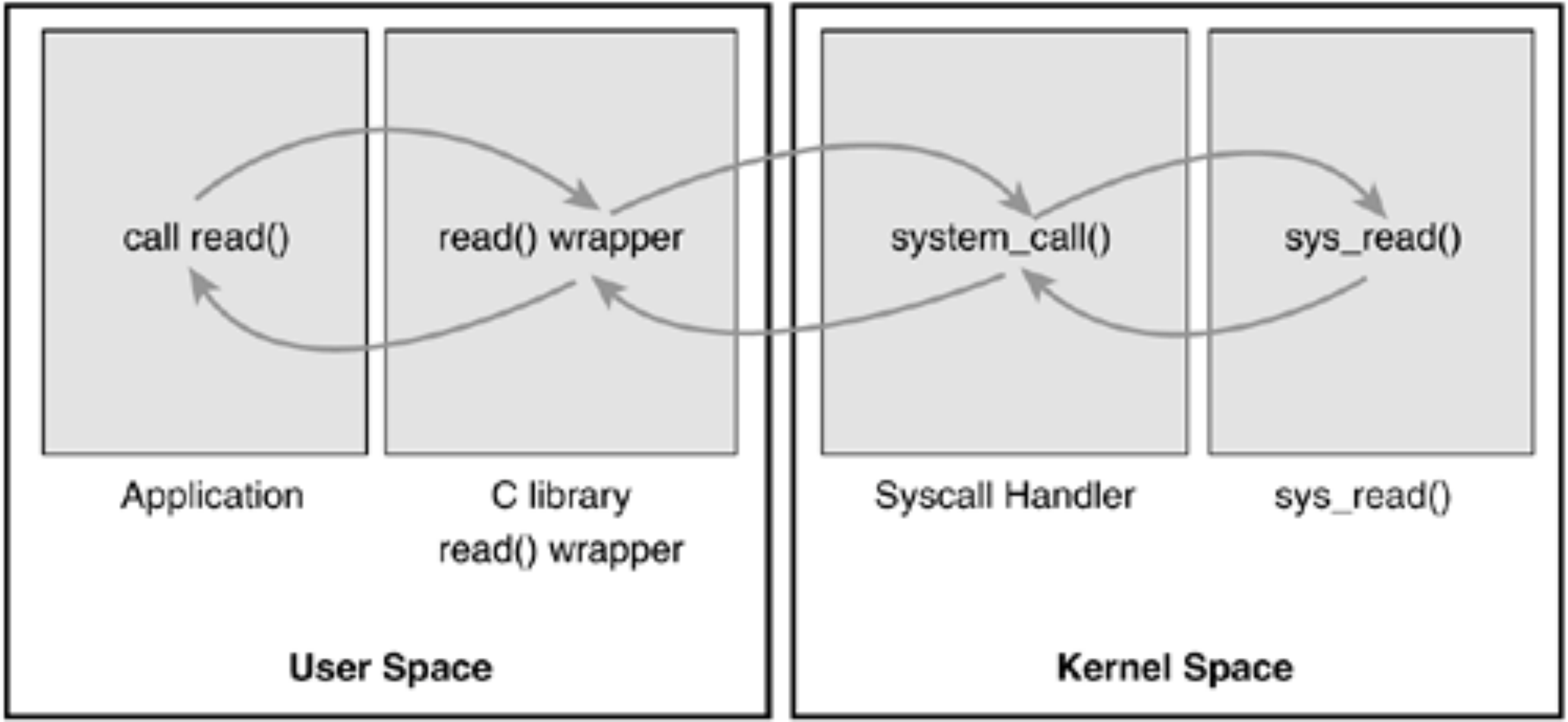


How are they called?

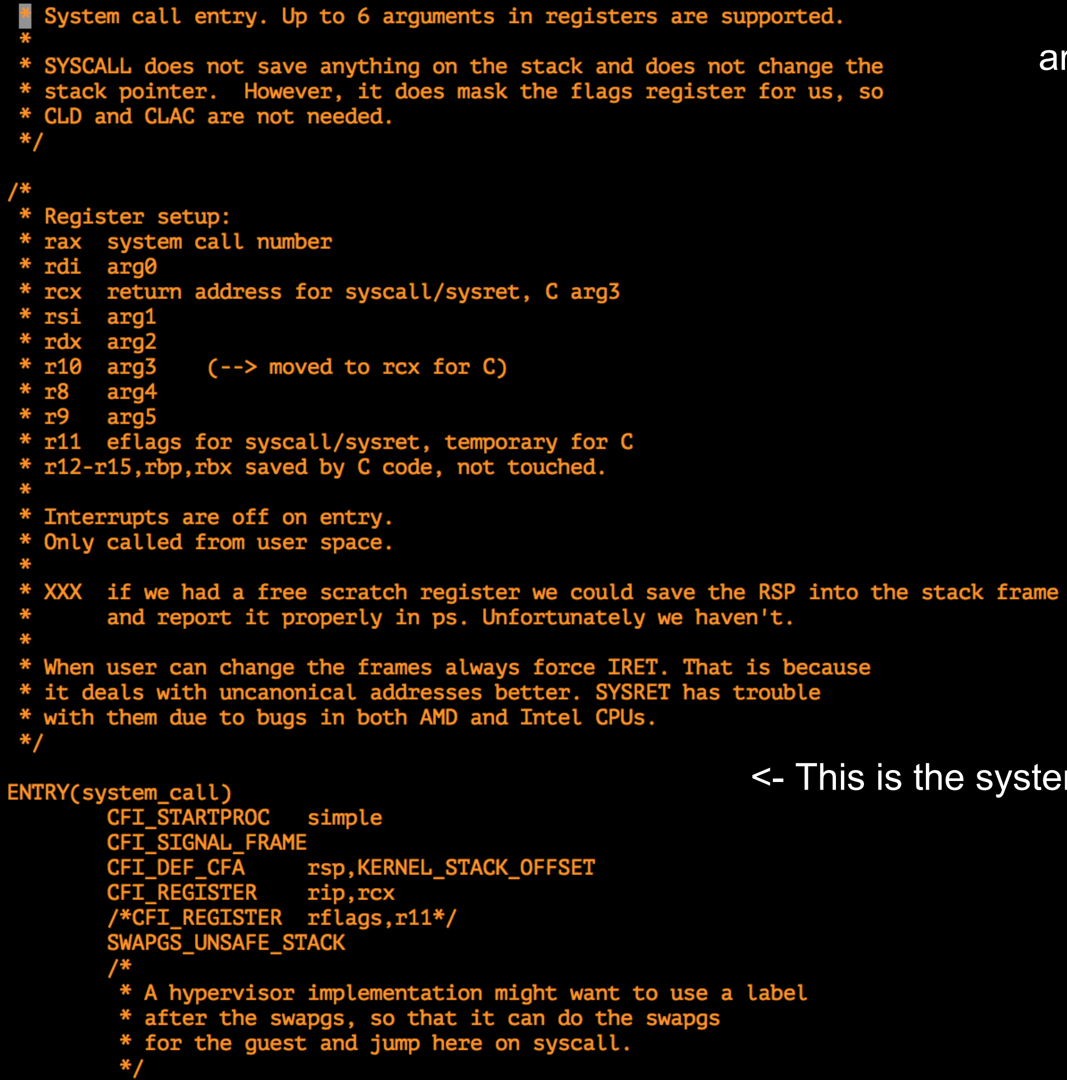
Called indirectly via a software interrupt/trap *(0x80),* the first argument is an integer so the function may be looked up in a syscall table

Because of this you don’t access them by name and they don’t need to be “included” in your code

```
int main() {
    syscall(3, fd, buf, count); // read
    return 0;
}
```


call read()
read() wrapper
system_call()
sys_read()
Application
C library
read() wrapper
Syscall Handler
sys_read()
User Space
Kernel Space

arch/x86/kernel/entry_64.S



```
 * System call entry. Up to 6 arguments in registers are supported.
 *
 * SYSCALL does not save anything on the stack and does not change the
 * stack pointer.  However, it does mask the flags register for us, so
 * CLD and CLAC are not needed.
 */

/*
 * Register setup:
 * rax  system call number
 * rdi  arg0
 * rcx  return address for syscall/sysret, C arg3
 * rsi  arg1
 * rdx  arg2
 * r10  arg3    (--> moved to rcx for C)
 * r8   arg4
 * r9   arg5
 * r11  eflags for syscall/sysret, temporary for C
 * r12-r15,rbp,rbx saved by C code, not touched.
 *
 * Interrupts are off on entry.
 * Only called from user space.
 *
 * XXX  if we had a free scratch register we could save the RSP into the stack frame
 *      and report it properly in ps. Unfortunately we haven't.
 *
 * When user can change the frames always force IRET. That is because
 * it deals with uncanonical addresses better. SYSRET has trouble
 * with them due to bugs in both AMD and Intel CPUs.
 */

ENTRY(system_call)
        CFI_STARTPROC   simple
        CFI_SIGNAL_FRAME
        CFI_DEF_CFA     rsp,KERNEL_STACK_OFFSET
        CFI_REGISTER    rip,rcx
        /*CFI_REGISTER  rflags,r11*/
        SWAPGS_UNSAFE_STACK
        /*
         * A hypervisor implementation might want to use a label
         * after the swapgs, so that it can do the swapgs
         * for the guest and jump here on syscall.
         */
```

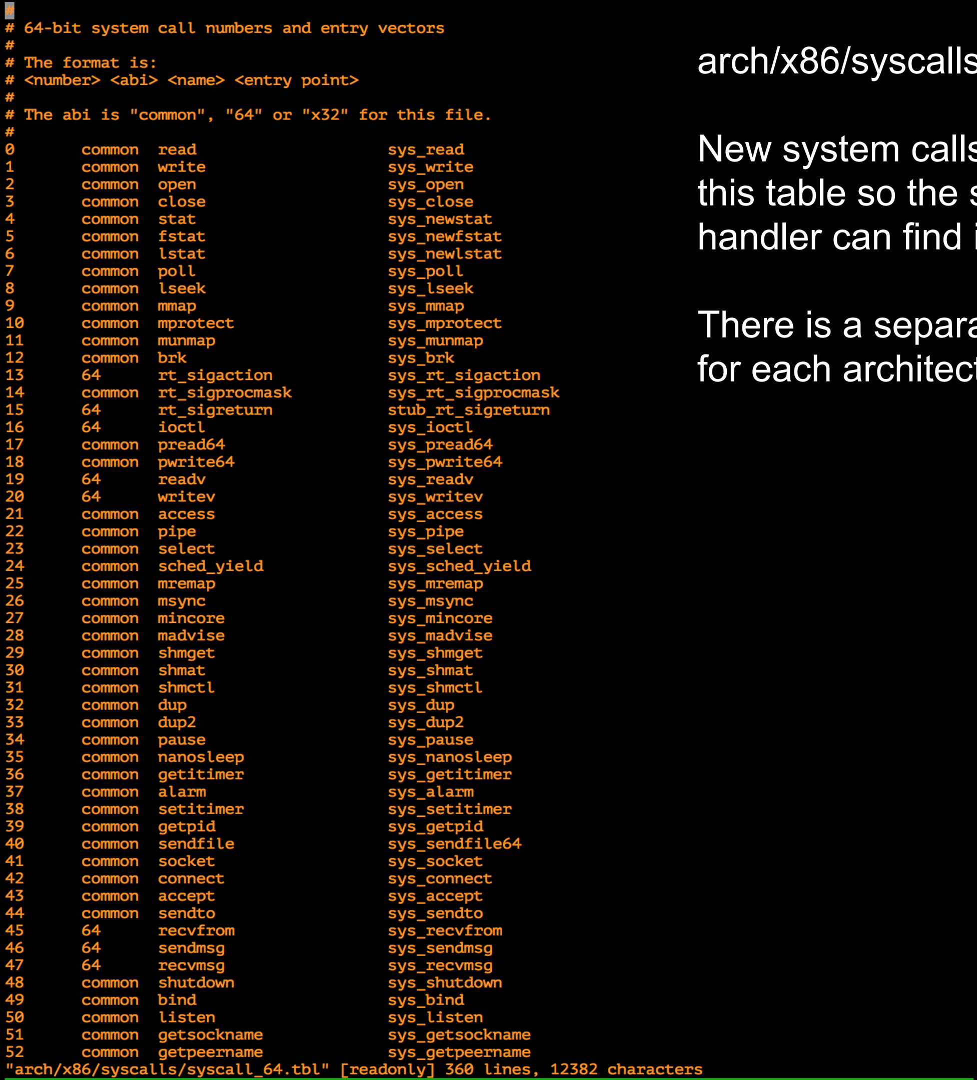
<- This is the system call interrupt handler.

How is a new syscall added?

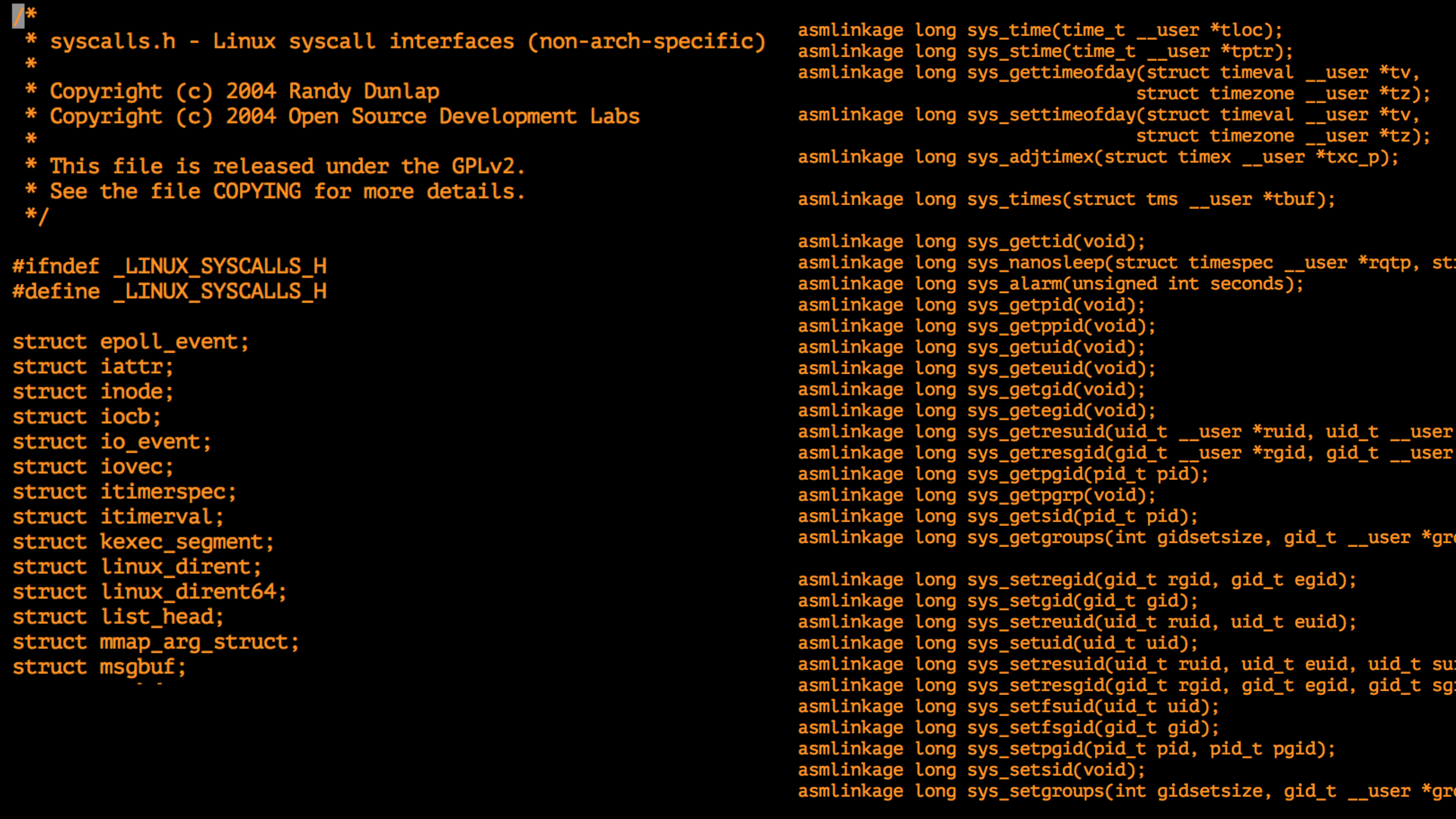
arch/x86/syscalls/syscall_64.tbl

New system calls must be added to this table so the system_call interrupt handler can find it.

There is a separate table of entries for each architecture.



```
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The abi is "common", "64" or "x32" for this file.
#
0       common  read                    sys_read
1       common  write                   sys_write
2       common  open                    sys_open
3       common  close                   sys_close
4       common  stat                    sys_newstat
5       common  fstat                   sys_newfstat
6       common  lstat                   sys_newlstat
7       common  poll                    sys_poll
8       common  lseek                   sys_lseek
9       common  mmap                    sys_mmap
10      common  mprotect                sys_mprotect
11      common  munmap                  sys_munmap
12      common  brk                     sys_brk
13      64      rt_sigaction            sys_rt_sigaction
14      common  rt_sigprocmask          sys_rt_sigprocmask
15      64      rt_sigreturn            stub_rt_sigreturn
16      64      ioctl                   sys_ioctl
17      common  pread64                 sys_pread64
18      common  pwrite64                sys_pwrite64
19      64      readv                   sys_readv
20      64      writev                  sys_writev
21      common  access                  sys_access
22      common  pipe                    sys_pipe
23      common  select                  sys_select
24      common  sched_yield             sys_sched_yield
25      common  mremap                  sys_mremap
26      common  msync                   sys_msync
27      common  mincore                 sys_mincore
28      common  madvise                 sys_madvise
29      common  shmget                  sys_shmget
30      common  shmat                   sys_shmat
31      common  shmctl                  sys_shmctl
32      common  dup                     sys_dup
33      common  dup2                    sys_dup2
34      common  pause                   sys_pause
35      common  nanosleep               sys_nanosleep
36      common  getitimer               sys_getitimer
37      common  alarm                   sys_alarm
38      common  setitimer               sys_setitimer
39      common  getpid                  sys_getpid
40      common  sendfile                sys_sendfile64
41      common  socket                  sys_socket
42      common  connect                 sys_connect
43      common  accept                  sys_accept
44      common  sendto                  sys_sendto
45      64      recvfrom                sys_recvfrom
46      64      sendmsg                 sys_sendmsg
47      64      recvmsg                 sys_recvmsg
48      common  shutdown                sys_shutdown
49      common  bind                    sys_bind
50      common  listen                  sys_listen
51      common  getsockname             sys_getsockname
52      common  getpeername             sys_getpeername
"arch/x86/syscalls/syscall_64.tbl" [readonly] 360 lines, 12382 characters
```



```
/*
 * syscalls.h - Linux syscall interfaces (non-arch-specific)
 *
 * Copyright (c) 2004 Randy Dunlap
 * Copyright (c) 2004 Open Source Development Labs
 *
 * This file is released under the GPLv2.
 * See the file COPYING for more details.
 */

#ifndef _LINUX_SYSCALLS_H
#define _LINUX_SYSCALLS_H

struct epoll_event;
struct iattr;
struct inode;
struct iocb;
struct io_event;
struct iovec;
struct itimerspec;
struct itimerval;
struct kexec_segment;
struct linux_dirent;
struct linux_dirent64;
struct list_head;
struct mmap_arg_struct;
struct msgbuf;
```

```
asmlinkage long sys_time(time_t __user *tloc);
asmlinkage long sys_stime(time_t __user *tptr);
asmlinkage long sys_gettimeofday(struct timeval __user *tv,
                                 struct timezone __user *tz);
asmlinkage long sys_settimeofday(struct timeval __user *tv,
                                 struct timezone __user *tz);
asmlinkage long sys_adjtimex(struct timex __user *txc_p);

asmlinkage long sys_times(struct tms __user *tbuf);

asmlinkage long sys_gettid(void);
asmlinkage long sys_nanosleep(struct timespec __user *rqtp, st
asmlinkage long sys_alarm(unsigned int seconds);
asmlinkage long sys_getpid(void);
asmlinkage long sys_getppid(void);
asmlinkage long sys_getuid(void);
asmlinkage long sys_geteuid(void);
asmlinkage long sys_getgid(void);
asmlinkage long sys_getegid(void);
asmlinkage long sys_getresuid(uid_t __user *ruid, uid_t __user
asmlinkage long sys_getresgid(gid_t __user *rgid, gid_t __user
asmlinkage long sys_getpgid(pid_t pid);
asmlinkage long sys_getpgrp(void);
asmlinkage long sys_getsid(pid_t pid);
asmlinkage long sys_getgroups(int gidsetsize, gid_t __user *gr

asmlinkage long sys_setregid(gid_t rgid, gid_t egid);
asmlinkage long sys_setgid(gid_t gid);
asmlinkage long sys_setreuid(uid_t ruid, uid_t euid);
asmlinkage long sys_setuid(uid_t uid);
asmlinkage long sys_setresuid(uid_t ruid, uid_t euid, uid_t su
asmlinkage long sys_setresgid(gid_t rgid, gid_t egid, gid_t sg
asmlinkage long sys_setfsuid(uid_t uid);
asmlinkage long sys_setfsgid(gid_t gid);
asmlinkage long sys_setpgid(pid_t pid, pid_t pgid);
asmlinkage long sys_setsid(void);
asmlinkage long sys_setgroups(int gidsetsize, gid_t __user *gr
```

Implementation

```
asmlinkage long sys_callcount(unsigned long * num) {
    ....
  }
```

asmlinkage is defined in <linux/linkage.h>; tells the compiler that the function will find its arguments on the CPU stack instead of in registers (as it normally would)

```
; making a syscall from user space in assembly
mov ebx, num ; address of num variable
mov     eax, 314         ; syscall number (sys_sethostname)
int 0x80         ; x86 call the kernel
```

When int 0x80 is called the cpu switches into kernel mode; the values of the registers are all saved to the cpu stack. So, this is how we can access the user space arguments from kernel mode...

Does it have to return a long?

Returning a long, or int in older kernels, allows you to handle errors, ex:

```
 if(copy_to_user(dst, &src, len))
     return -EFAULT
```

I wrote a void syscall; it still works!

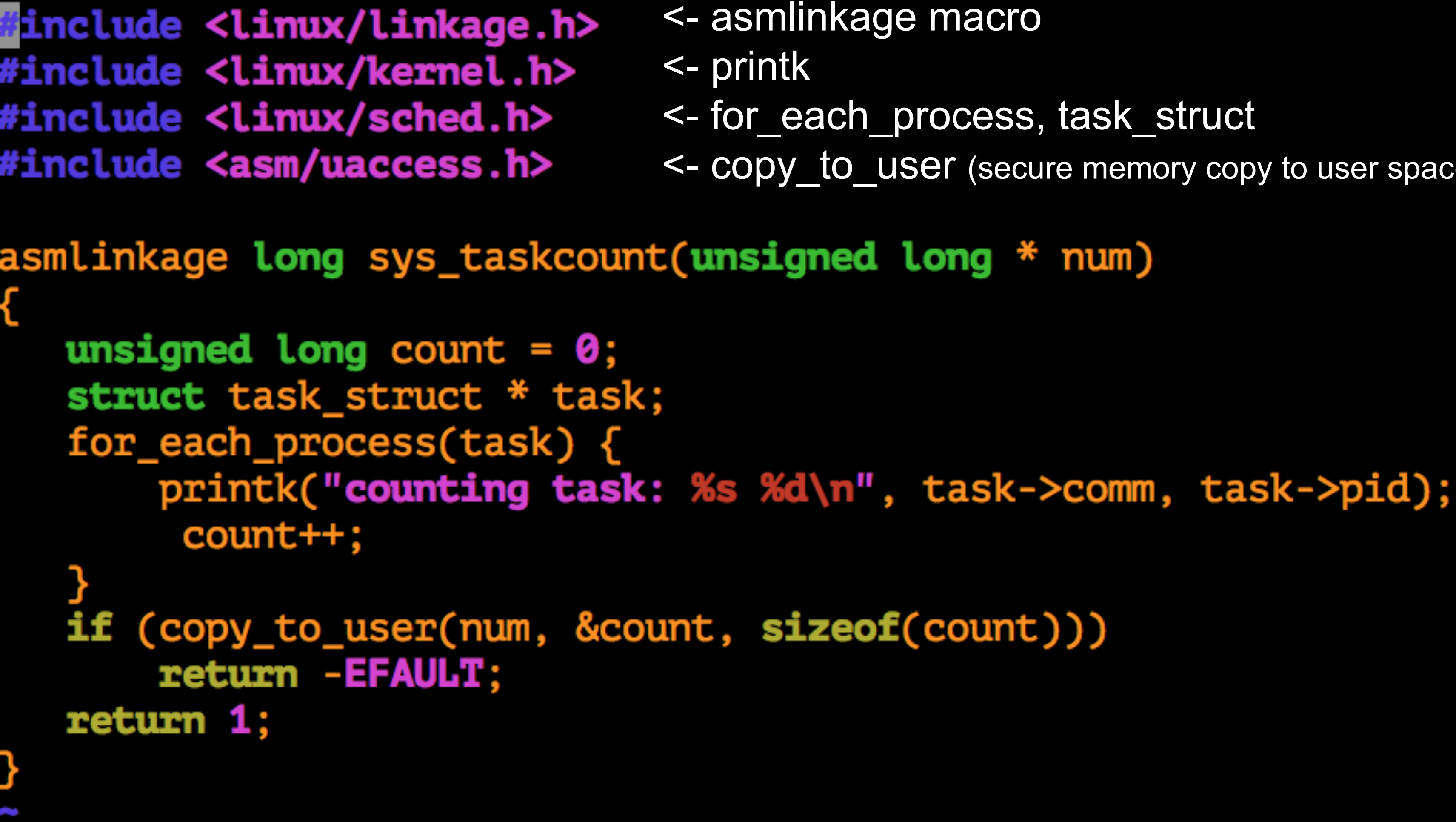
Some Security Considerations

* Processes must not be able to trick the kernel into reading data in kernel-space on their behalf.

* The process must not be able to trick the kernel into reading someone else's data.

* The process must not be able to bypass memory access restrictions. rwxrwxrwx

<asm/uaccess.h> has helper functions to address these issues: copy_to_user, copy_from_user, etc….



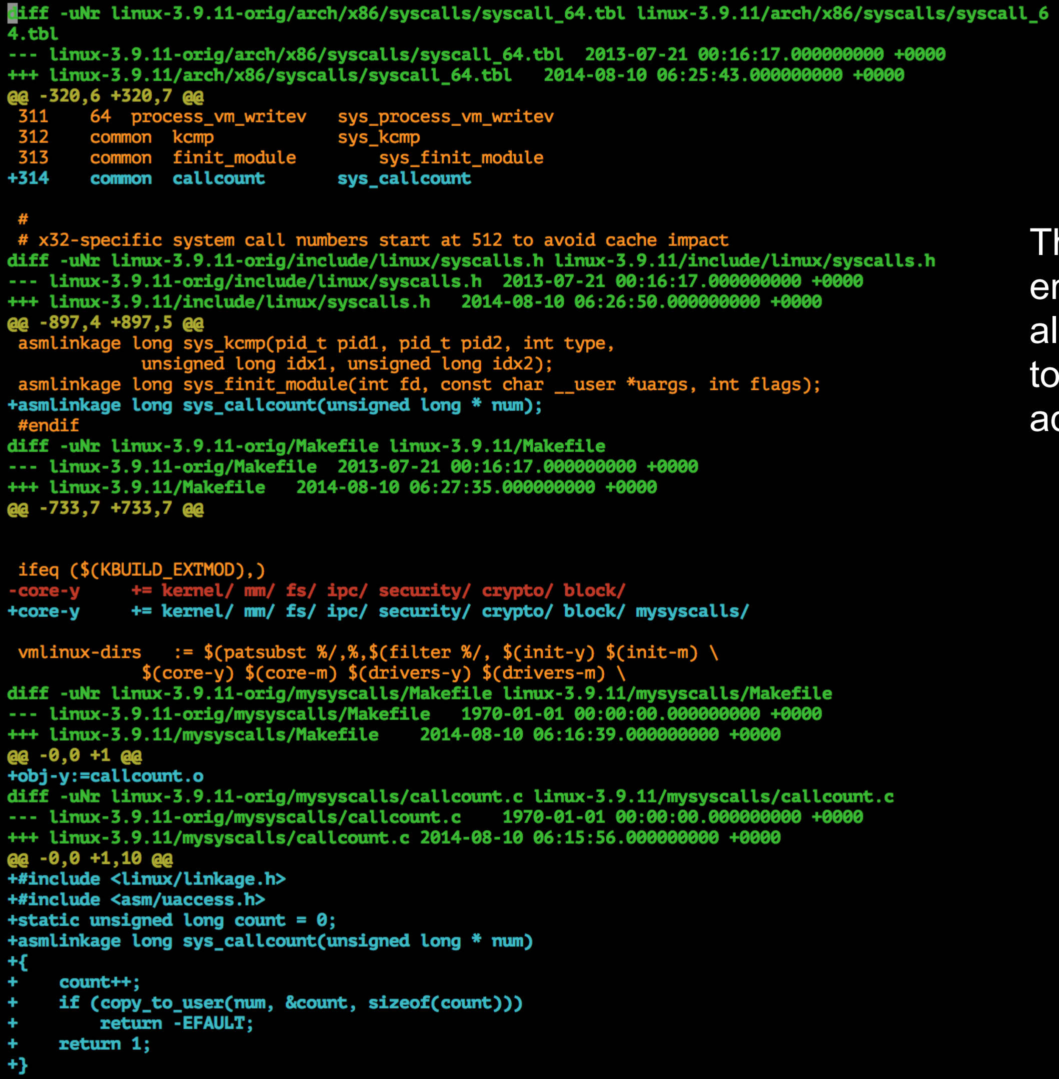
```
#include <linux/linkage.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <asm/uaccess.h>

asmlinkage long sys_taskcount(unsigned long * num)
{
   unsigned long count = 0;
   struct task_struct * task;
   for_each_process(task) {
       printk("counting task: %s %d\n", task->comm, task->pid);
        count++;
   }
   if (copy_to_user(num, &count, sizeof(count)))
       return -EFAULT;
   return 1;
}
~
```

<- asmlinkage macro

<- printk

<- for_each_process, task_struct

<- copy_to_user (secure memory copy to user space)



```
diff -uNr linux-3.9.11-orig/arch/x86/syscalls/syscall_64.tbl linux-3.9.11/arch/x86/syscalls/syscall_6
4.tbl
--- linux-3.9.11-orig/arch/x86/syscalls/syscall_64.tbl  2013-07-21 00:16:17.000000000 +0000
+++ linux-3.9.11/arch/x86/syscalls/syscall_64.tbl   2014-08-10 06:25:43.000000000 +0000
@@ -320,6 +320,7 @@
 311    64   process_vm_writev   sys_process_vm_writev
 312    common  kcmp                sys_kcmp
 313    common  finit_module            sys_finit_module
+314    common  callcount           sys_callcount
 
 #
 # x32-specific system call numbers start at 512 to avoid cache impact
diff -uNr linux-3.9.11-orig/include/linux/syscalls.h linux-3.9.11/include/linux/syscalls.h
--- linux-3.9.11-orig/include/linux/syscalls.h  2013-07-21 00:16:17.000000000 +0000
+++ linux-3.9.11/include/linux/syscalls.h   2014-08-10 06:26:50.000000000 +0000
@@ -897,4 +897,5 @@
 asmlinkage long sys_kcmp(pid_t pid1, pid_t pid2, int type,
             unsigned long idx1, unsigned long idx2);
 asmlinkage long sys_finit_module(int fd, const char __user *uargs, int flags);
+asmlinkage long sys_callcount(unsigned long * num);
 #endif
diff -uNr linux-3.9.11-orig/Makefile linux-3.9.11/Makefile
--- linux-3.9.11-orig/Makefile  2013-07-21 00:16:17.000000000 +0000
+++ linux-3.9.11/Makefile   2014-08-10 06:27:35.000000000 +0000
@@ -733,7 +733,7 @@
 
 
 ifeq ($(KBUILD_EXTMOD),)
-core-y     += kernel/ mm/ fs/ ipc/ security/ crypto/ block/
+core-y     += kernel/ mm/ fs/ ipc/ security/ crypto/ block/ mysyscalls/
 
 vmlinux-dirs   := $(patsubst %/,%,$(filter %/, $(init-y) $(init-m) \
             $(core-y) $(core-m) $(drivers-y) $(drivers-m) \
diff -uNr linux-3.9.11-orig/mysyscalls/Makefile linux-3.9.11/mysyscalls/Makefile
--- linux-3.9.11-orig/mysyscalls/Makefile   1970-01-01 00:00:00.000000000 +0000
+++ linux-3.9.11/mysyscalls/Makefile    2014-08-10 06:16:39.000000000 +0000
@@ -0,0 +1 @@
+obj-y:=callcount.o
diff -uNr linux-3.9.11-orig/mysyscalls/callcount.c linux-3.9.11/mysyscalls/callcount.c
--- linux-3.9.11-orig/mysyscalls/callcount.c    1970-01-01 00:00:00.000000000 +0000
+++ linux-3.9.11/mysyscalls/callcount.c 2014-08-10 06:15:56.000000000 +0000
@@ -0,0 +1,10 @@
+#include <linux/linkage.h>
+#include <asm/uaccess.h>
+static unsigned long count = 0;
+asmlinkage long sys_callcount(unsigned long * num)
+{
+    count++;
+    if (copy_to_user(num, &count, sizeof(count)))
+        return -EFAULT;
+    return 1;
+}
```

This patch
encapsulates
all of the steps necessary
to
add a system call.

If you wish to add a system call...

you must build the kernel

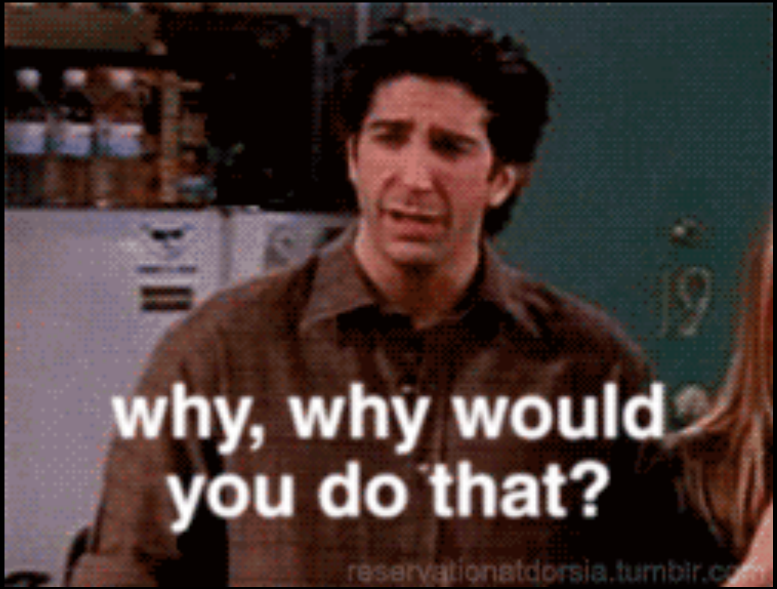
Other Reasons To Build A Kernel

Compile with non-default flags to add things like high memory support for special use cases

Optimize boot time by removing unnecessary modules

Run a development kernel

Learn more about linux kernels



Steps To Build The Kernel

```
$ wget http://www.kernel.org/pub/linux/kernel/v3.0/linux-3.9.tar.bz2
$ tar xvzf linux-3.9.tar.gz && cd linux-3.9
```

using our current kernel config as a template, fill in the missing options with a prompt, and generate a new .config

```
$  cp /boot/config-`uname -r` .config && make oldconfig
```

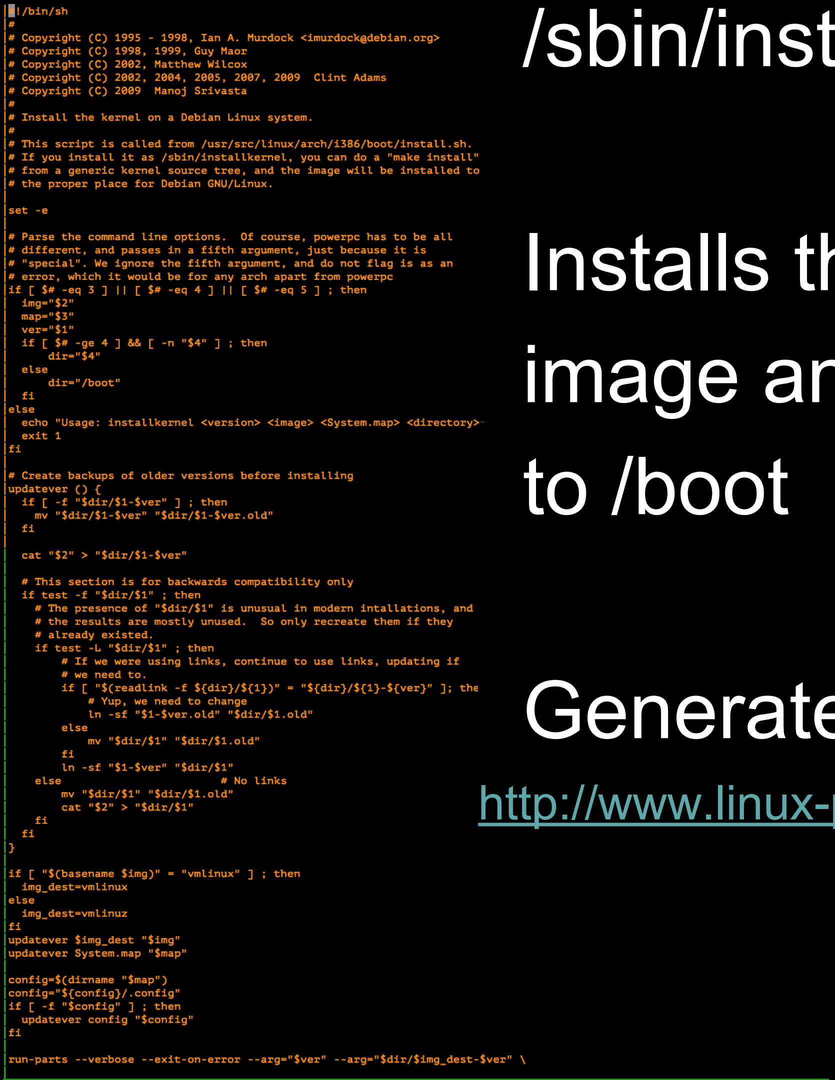
compile kernel image (vmlinuz), compile modules, install modules to /lib/modules/3.9.11, calls /sbin/installkernel

```
$ make -j<num_of_cpu_cores + 1> && make modules && make modules_install install
```

search /, /boot, and /lib/modules for kernel versions and add them to the bootloader config

```
$ update-grub2 || update-grub || update-burg
```

/sbin/installkernel

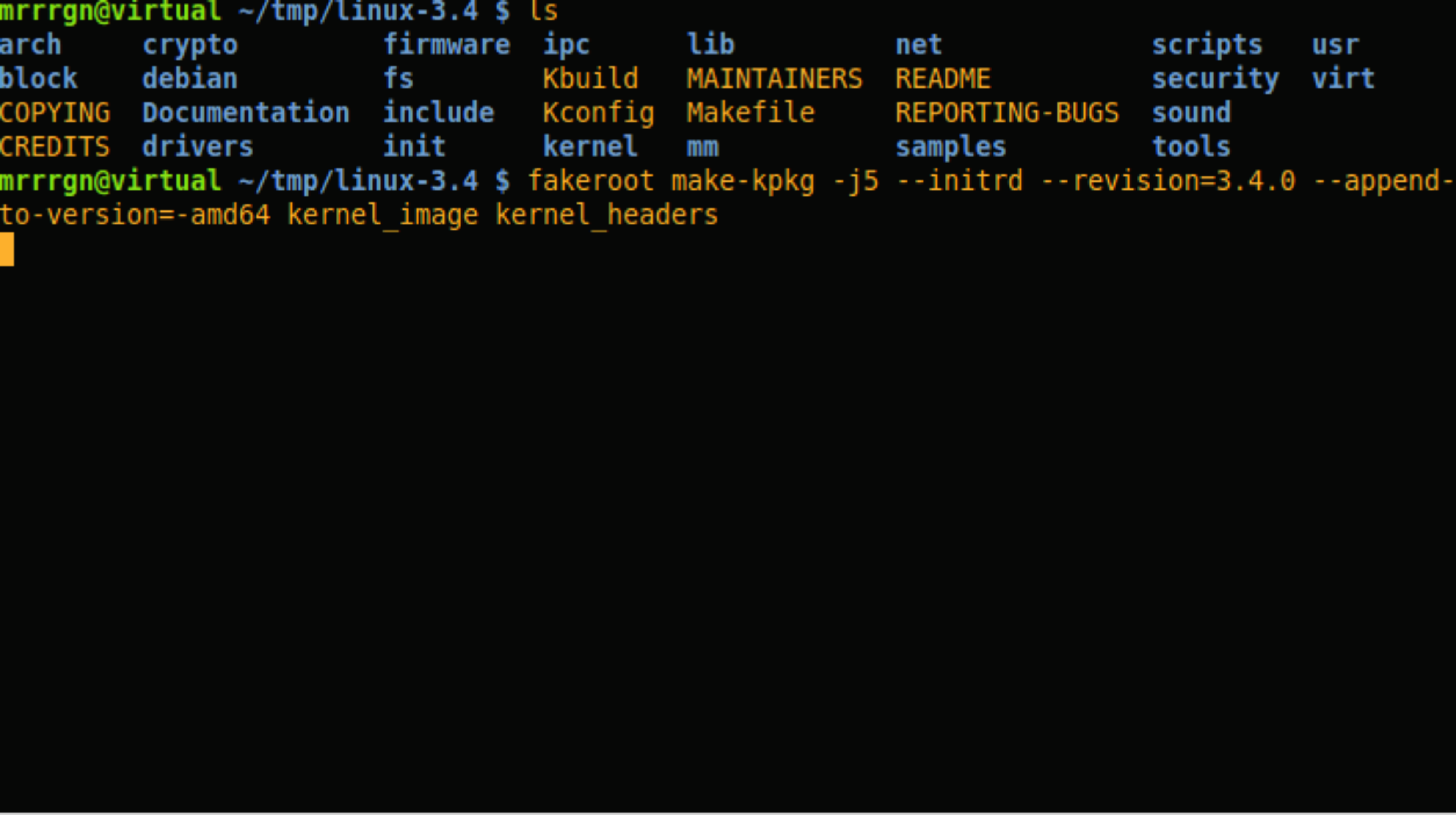


```
#!/bin/sh
#
# Copyright (C) 1995 - 1998, Ian A. Murdock <imurdock@debian.org>
# Copyright (C) 1998, 1999, Guy Maor
# Copyright (C) 2002, Matthew Wilcox
# Copyright (C) 2002, 2004, 2005, 2007, 2009  Clint Adams
# Copyright (C) 2009  Manoj Srivasta
#
# Install the kernel on a Debian Linux system.
#
# This script is called from /usr/src/linux/arch/i386/boot/install.sh.
# If you install it as /sbin/installkernel, you can do a "make install"
# from a generic kernel source tree, and the image will be installed to
# the proper place for Debian GNU/Linux.

set -e

# Parse the command line options.  Of course, powerpc has to be all
# different, and passes in a fifth argument, just because it is
# "special". We ignore the fifth argument, and do not flag is as an
# error, which it would be for any arch apart from powerpc
if [ $# -eq 3 ] || [ $# -eq 4 ] || [ $# -eq 5 ] ; then
  img="$2"
  map="$3"
  ver="$1"
  if [ $# -ge 4 ] && [ -n "$4" ] ; then
      dir="$4"
  else
      dir="/boot"
  fi
else
  echo "Usage: installkernel <version> <image> <System.map> <directory>
  exit 1
fi

# Create backups of older versions before installing
updatever () {
  if [ -f "$dir/$1-$ver" ] ; then
    mv "$dir/$1-$ver" "$dir/$1-$ver.old"
  fi

  cat "$2" > "$dir/$1-$ver"

  # This section is for backwards compatibility only
  if test -f "$dir/$1" ; then
    # The presence of "$dir/$1" is unusual in modern intallations, and
    # the results are mostly unused.  So only recreate them if they
    # already existed.
    if test -L "$dir/$1" ; then
        # If we were using links, continue to use links, updating if
        # we need to.
        if [ "$(readlink -f ${dir}/${1})" = "${dir}/${1}-${ver}" ]; the
            # Yup, we need to change
            ln -sf "$1-$ver.old" "$dir/$1.old"
        else
            mv "$dir/$1" "$dir/$1.old"
        fi
        ln -sf "$1-$ver" "$dir/$1"
    else                        # No links
        mv "$dir/$1" "$dir/$1.old"
        cat "$2" > "$dir/$1"
    fi
  fi
}

if [ "$(basename $img)" = "vmlinux" ] ; then
  img_dest=vmlinux
else
  img_dest=vmlinuz
fi
updatever $img_dest "$img"
updatever System.map "$map"

config=$(dirname "$map")
config="${config}/.config"
if [ -f "$config" ] ; then
  updatever config "$config"
fi

run-parts --verbose --exit-on-error --arg="$ver" --arg="$dir/$img_dest-$ver" \
```

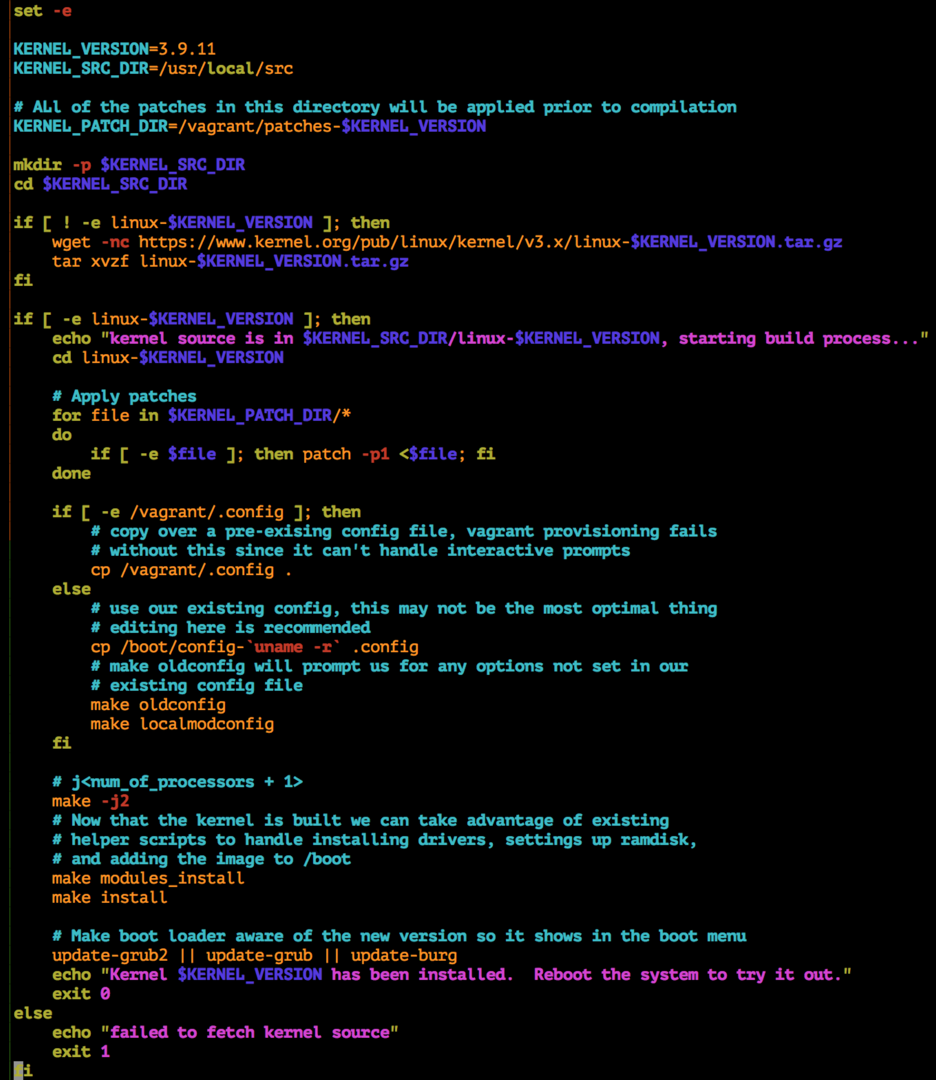
Installs the vmlinuz image and .config to /boot

Generates initrd

http://www.linux-poetry.com/14/



```
mrrrgn@virtual ~/tmp/linux-3.4 $ ls
arch     crypto         firmware  ipc      lib          net             scripts   usr
block    debian         fs        Kbuild   MAINTAINERS  README          security  virt
COPYING  Documentation  include   Kconfig  Makefile     REPORTING-BUGS  sound
CREDITS  drivers        init      kernel   mm           samples         tools
mrrrgn@virtual ~/tmp/linux-3.4 $ fakeroot make-kpkg -j5 --initrd --revision=3.4.0 --append-
to-version=-amd64 kernel_image kernel_headers
```



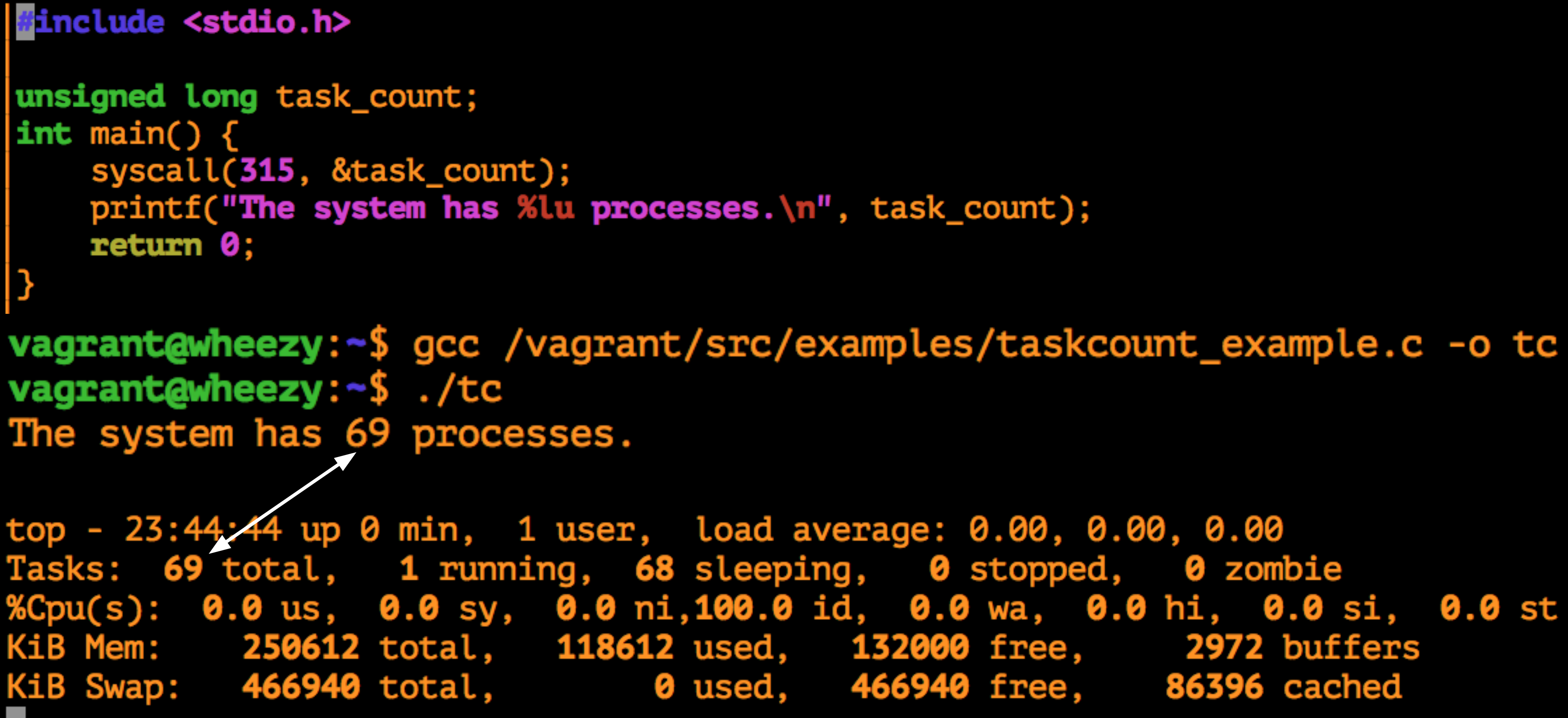
```
set -e

KERNEL_VERSION=3.9.11
KERNEL_SRC_DIR=/usr/local/src

# All of the patches in this directory will be applied prior to compilation
KERNEL_PATCH_DIR=/vagrant/patches-$KERNEL_VERSION

mkdir -p $KERNEL_SRC_DIR
cd $KERNEL_SRC_DIR

if [ ! -e linux-$KERNEL_VERSION ]; then
    wget -nc https://www.kernel.org/pub/linux/kernel/v3.x/linux-$KERNEL_VERSION.tar.gz
    tar xvzf linux-$KERNEL_VERSION.tar.gz
fi

if [ -e linux-$KERNEL_VERSION ]; then
    echo "kernel source is in $KERNEL_SRC_DIR/linux-$KERNEL_VERSION, starting build process..."
    cd linux-$KERNEL_VERSION

    # Apply patches
    for file in $KERNEL_PATCH_DIR/*
    do
        if [ -e $file ]; then patch -p1 <$file; fi
    done

    if [ -e /vagrant/.config ]; then
        # copy over a pre-exising config file, vagrant provisioning fails
        # without this since it can't handle interactive prompts
        cp /vagrant/.config .
    else
        # use our existing config, this may not be the most optimal thing
        # editing here is recommended
        cp /boot/config-`uname -r` .config
        # make oldconfig will prompt us for any options not set in our
        # existing config file
        make oldconfig
        make localmodconfig
    fi

    # j<num_of_processors + 1>
    make -j2
    # Now that the kernel is built we can take advantage of existing
    # helper scripts to handle installing drivers, settings up ramdisk,
    # and adding the image to /boot
    make modules_install
    make install

    # Make boot loader aware of the new version so it shows in the boot menu
    update-grub2 || update-grub || update-burg
    echo "Kernel $KERNEL_VERSION has been installed.  Reboot the system to try it out."
    exit 0
else
    echo "failed to fetch kernel source"
    exit 1
fi
```

upgrade-kernel.sh

A bash script which downloads the 3.9.11 kernel, compiles it, and installs it on a system.

https://github.com/mrrrgn/syscall-3.9/blob/master/upgrade-kernel.sh

Let’s test our task count system call



```c
#include <stdio.h>

unsigned long task_count;
int main() {
    syscall(315, &task_count);
    printf("The system has %lu processes.\n", task_count);
    return 0;
}
```

```
vagrant@wheezy:~$ gcc /vagrant/src/examples/taskcount_example.c -o tc
vagrant@wheezy:~$ ./tc
The system has 69 processes.
```

```
top - 23:44:44 up 0 min,  1 user,  load average: 0.00, 0.00, 0.00
Tasks:  69 total,   1 running,  68 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:    250612 total,   118612 used,   132000 free,     2972 buffers
KiB Swap:   466940 total,        0 used,   466940 free,    86396 cached
```

Win! Now we should roll our own calls for all the things….

It’s better to write modules:

:(Changing system calls will break userspace programs

:(System calls need to be supported for each architecture

:(Changing system calls requires compiling the kernel

:) Modules can be loaded and unloaded whenevs

:) Old versions can be loaded by sysops if necessary

:) Development moves faster

:) More likely to be accepted by maintainers

What have we learned?

* System calls are made by trapping into the kernel where the system_call interrupt handler looks up the function it should run (in a table)

* System calls use a neat compiler macro to get arguments from userspace.

* It’s easy to write highly exploitable kernel code; but there are helper functions that will make it more secure.

* In general writing modules is a better way to encapsulate kernel mode logic.