SYSCALLS

Feel the magic, hear the roar

The story so far

- What we can do now:
  - Kernel configuration
  - Kernel compilation
  - Kernel booting
  - Kernel Hacking
  - Kernel Patching
  - Kernel profiling
  - Kernel Debugging
- What else could we do to squeeze some more awesomeness out of our kernel?

System calls!!!

So… Who can tell me what a system call is?

- You!
- Yeah, you!
- The blonde kid smirking in the back
- Yeah
- The one behind the kid with the glasses
- What can you tell us about system calls?

Srsly though

- A system call (we'll just go ahead and call them "syscalls" from here on out) is the way a user-level program asks the OS to do something for it.
- This can be very useful as the OS may have access to resources unavailable to the user
- Linux has over 300 different system calls

Big picture

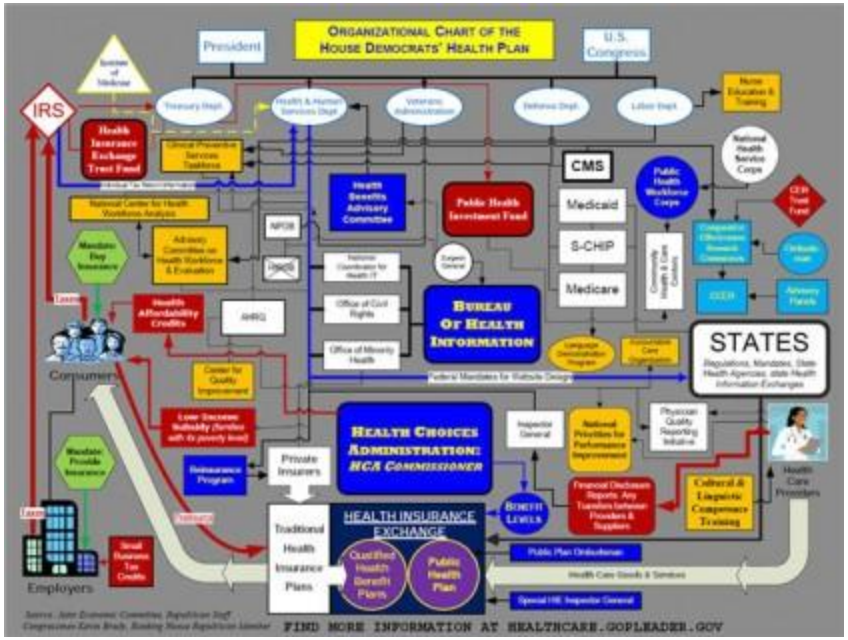
- The user level program, which operates on the lowest privilege level, requires a service (I/O, IPC, etc)
- It requests the service from the OS via a system call
- If allowed, the system obtains a higher privilege level, the processor jumps to a new address and starts executing the code there.
- After this finishes running, the original privilege level is returned to and control is relinquished back to the userspace process

Remind me again why I need to go through this hell?

- System calls provide another level of abstraction between the user and the hardware.
- There are two main reasons this is very good
- Any ideas?

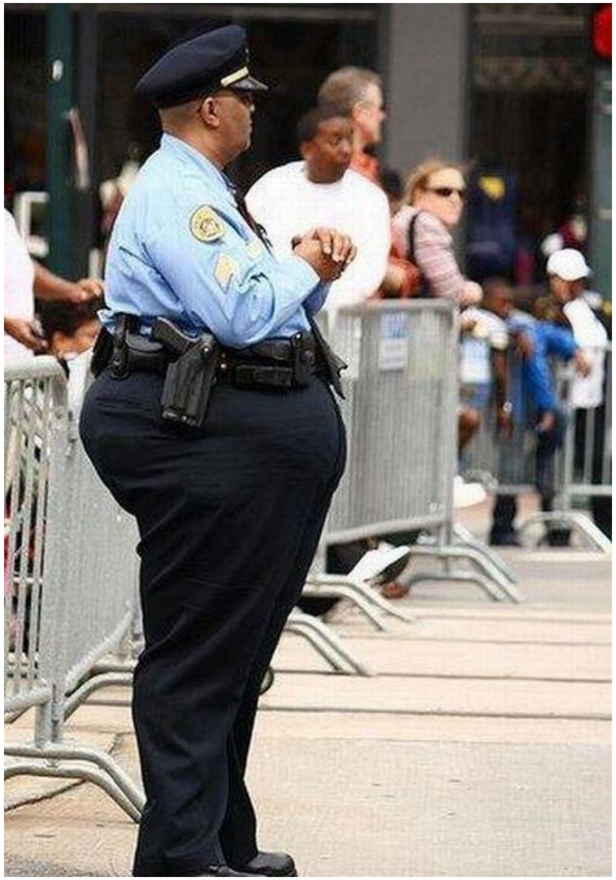
Complexity

- It spares the programmer the need to deal with the complexity a syscall must address



Security

- The computer has a chance to evaluate a request before it reaches the HW and potentially messes something up



Then and now

- Once upon a time (kernel 2.5 and below), to generate syscalls Linux used the “int 0x80” assembly instruction. The syscall number was placed in the EAX register and then interrupt 0x80 was executed
- Today, CISC architectures (like x86) use one of two “fast” control transfer mechanisms (one developed by AMD and the other by Intel) by which much of the interrupt overhead can be avoided.

A closer look

- After the system call is generated (whether by the "int" instruction or some new Intel/AMD technology) the processor jumps to a set of assembly instructions called a "syscall handler"
- The syscall hander saves the kernel context (Kernel mode register contents), calls the system call service routine and then returns the kernel context before returning to user mode.

So what do I need to get my system callin' on?

- We shall go over the following steps during this lecture:
  - Registering your new system call
  - Integrating its make file in the main Make hierarchy
  - Coding your syscall
  - Calling your syscall from a user space driver

All paths lead to Rome

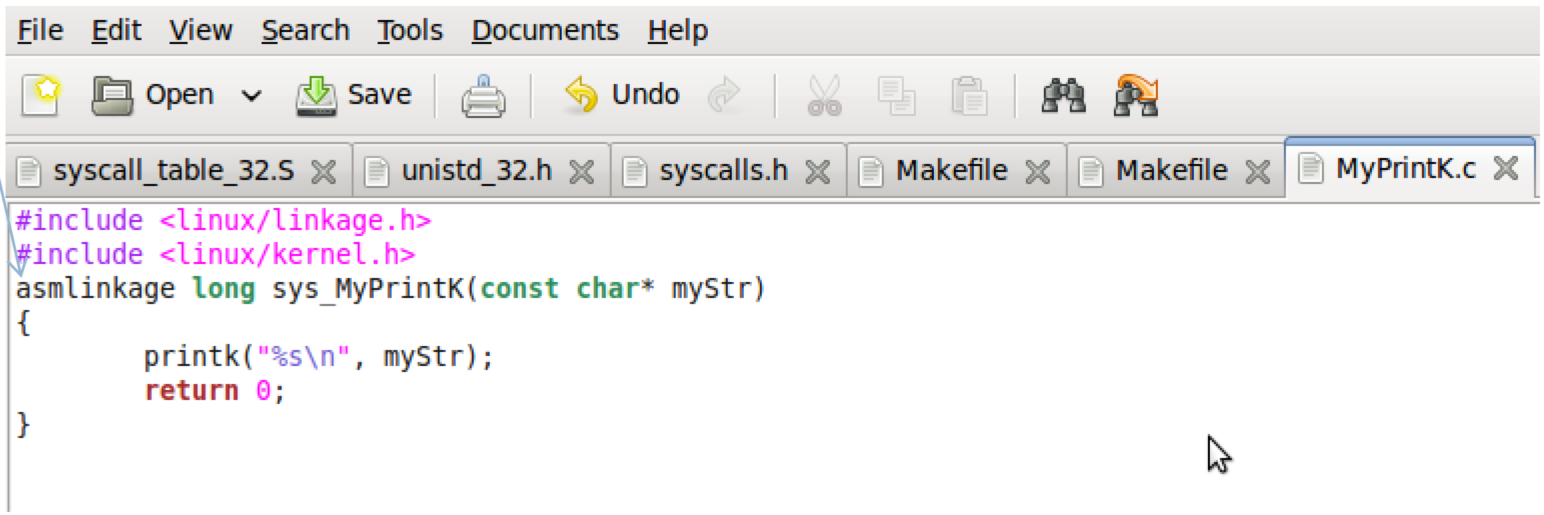
- There is more than one way to implement a syscall (the implementation could be made in an existing file, for example)
- We have put together the most generic one we could think of
- Feel free to improve on our technique

(1) Write your syscall

- Chose a directory within the kernel source (we have chosen the kernel's root directory in our example)
- Create a new directory where you shall store your syscall's implementation and Makefile
- Write your syscall's source (what you want it to do)

The system call implementation

The asmlinkage modifier tells the method to look for its arguments on the kernel stack



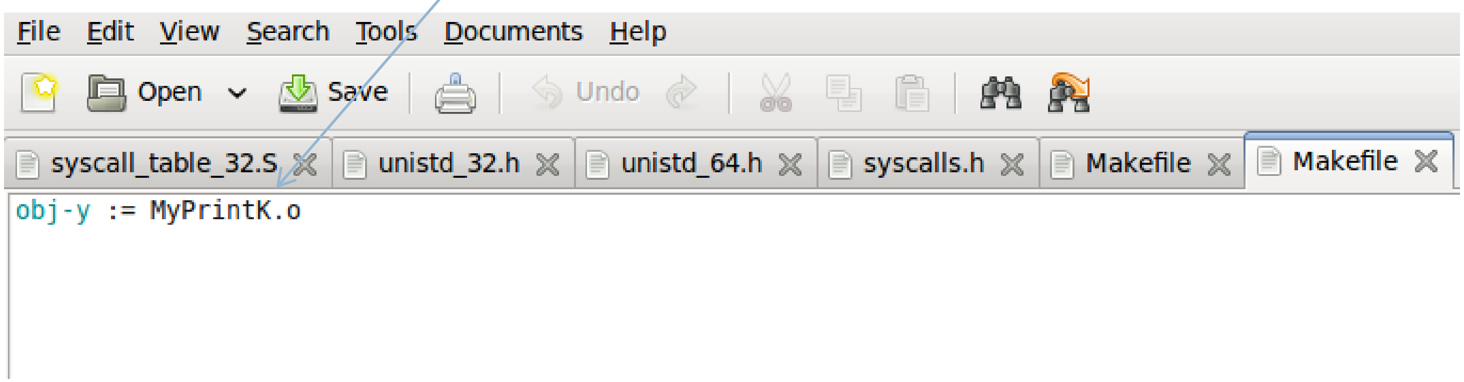

```c
#include <linux/linkage.h>
#include <linux/kernel.h>
asmlinkage long sys_MyPrintK(const char* myStr)
{
        printk("%s\n", myStr);
        return 0;
}
```

(1) Write your syscall (part 2)

- Now write up a make file for your system call
- Also, make sure that the kernel root Makefile triggers your system call Makefile so that you don't have to take care of it separately

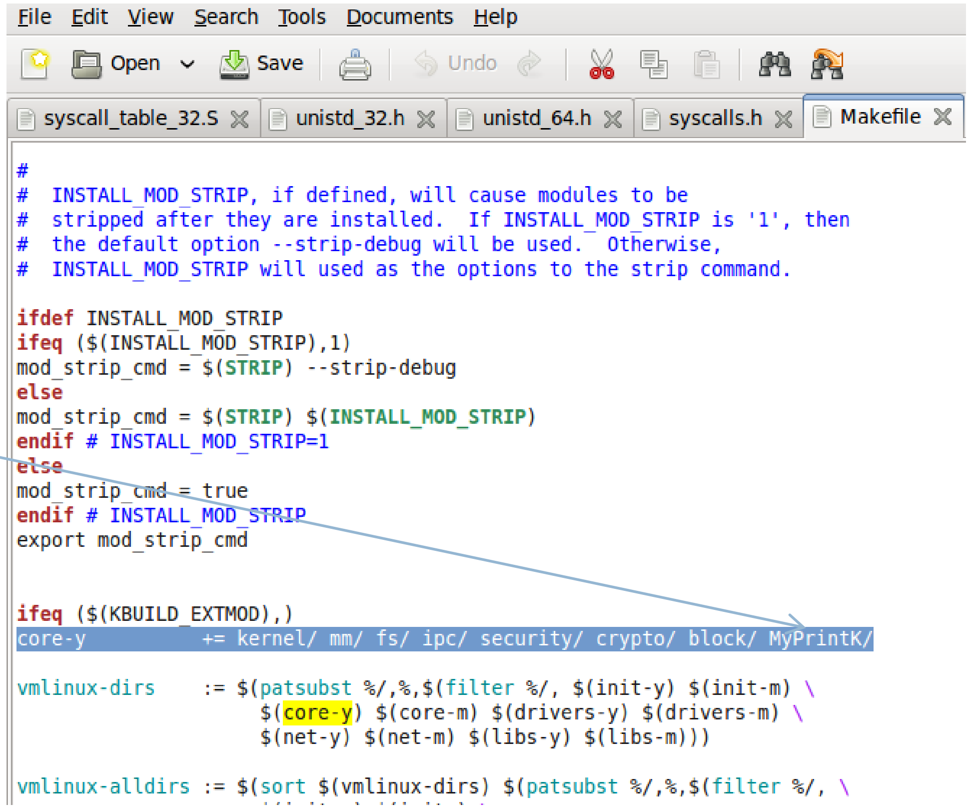
The Makefile

This flag ensures that your system call code is compiled with the main Makefile call



The root directory Makefile

Add the new directory that contains your system call




```
#
#  INSTALL_MOD_STRIP, if defined, will cause modules to be
#  stripped after they are installed.  If INSTALL_MOD_STRIP is '1', then
#  the default option --strip-debug will be used.  Otherwise,
#  INSTALL_MOD_STRIP will used as the options to the strip command.

ifdef INSTALL_MOD_STRIP
ifeq ($(INSTALL_MOD_STRIP),1)
mod_strip_cmd = $(STRIP) --strip-debug
else
mod_strip_cmd = $(STRIP) $(INSTALL_MOD_STRIP)
endif # INSTALL_MOD_STRIP=1
else
mod_strip_cmd = true
endif # INSTALL_MOD_STRIP
export mod_strip_cmd


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

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

vmlinux-alldirs := $(sort $(vmlinux-dirs) $(patsubst %/,%,$(filter %/, \
```

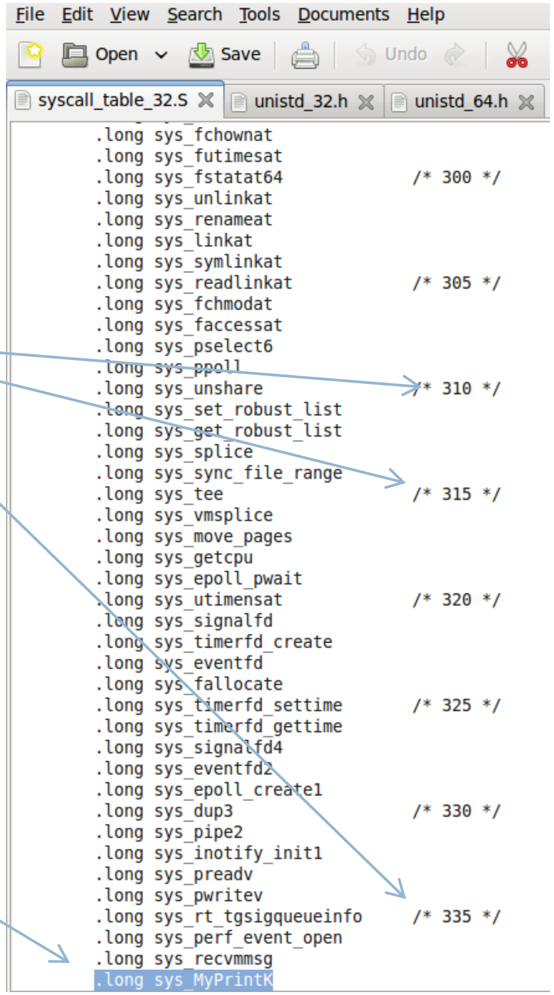
(2) Register your syscall

- We now need to register the syscall. This is done over several files so rack 'em up:
  - /arch/x86/kernel/*syscall_table_32.S*
  - include/linux/*syscalls.h*
  - /arch/x86/include/asm/*unistd_32.h*

System call registration I

These numbers keep track of what number each syscall has

Write your syscall name on the last line



System call registration II

File Edit View Search Tools Documents Help

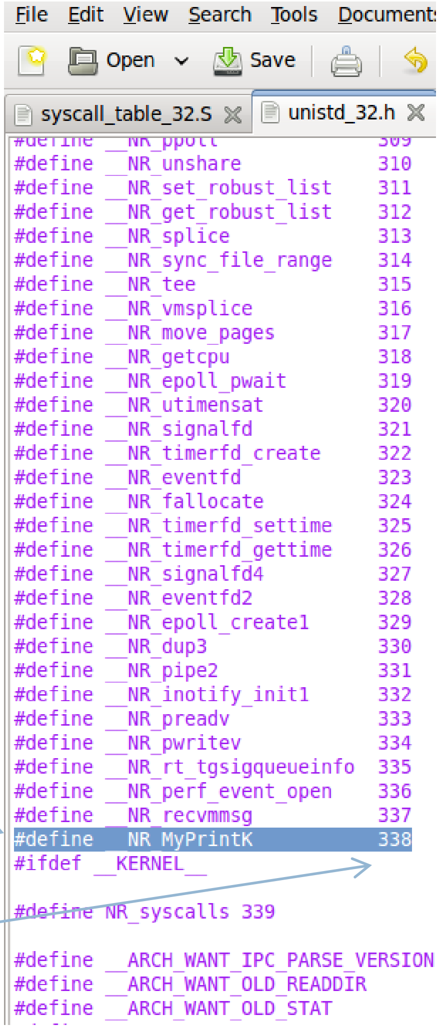
Open Save Undo

syscall_table_32.S | unistd_32.h | unistd_64.h | *syscalls.h

```c
asmlinkage long sys_sync_file_range2(int fd, unsigned int flags,
                                     loff_t offset, loff_t nbytes);
asmlinkage long sys_get_robust_list(int pid,
                                    struct robust_list_head __user * __user *head_ptr,
                                    size_t __user *len_ptr);
asmlinkage long sys_set_robust_list(struct robust_list_head __user *head,
                                    size_t len);
asmlinkage long sys_getcpu(unsigned __user *cpu, unsigned __user *node, struct getcpu_cache __user *cache);
asmlinkage long sys_signalfd(int ufd, sigset_t __user *user_mask, size_t sizemask);
asmlinkage long sys_signalfd4(int ufd, sigset_t __user *user_mask, size_t sizemask, int flags);
asmlinkage long sys_timerfd_create(int clockid, int flags);
asmlinkage long sys_timerfd_settime(int ufd, int flags,
                                    const struct itimerspec __user *utmr,
                                    struct itimerspec __user *otmr);
asmlinkage long sys_timerfd_gettime(int ufd, struct itimerspec __user *otmr);
asmlinkage long sys_eventfd(unsigned int count);
asmlinkage long sys_eventfd2(unsigned int count, int flags);
asmlinkage long sys_fallocate(int fd, int mode, loff_t offset, loff_t len);
asmlinkage long sys_old_readdir(unsigned int, struct old_linux_dirent __user *, unsigned int);
asmlinkage long sys_pselect6(int, fd_set __user *, fd_set __user *,
                             fd_set __user *, struct timespec __user *,
                             void __user *);
asmlinkage long sys_ppoll(struct pollfd __user *, unsigned int,
                          struct timespec __user *, const sigset_t __user *,
                          size_t);

int kernel_execve(const char *filename, char *const argv[], char *const envp[]);


asmlinkage long sys_perf_event_open(
                struct perf_event_attr __user *attr_uptr,
                pid_t pid, int cpu, int group_fd, unsigned long flags);

asmlinkage long sys_mmap_pgoff(unsigned long addr, unsigned long len,
                        unsigned long prot, unsigned long flags,
                        unsigned long fd, unsigned long pgoff);
asmlinkage long sys_old_mmap(struct mmap_arg_struct __user *arg);

asmlinkage long sys_MyPrintK(const char* myStr);

#endif
```

Declare your system call here

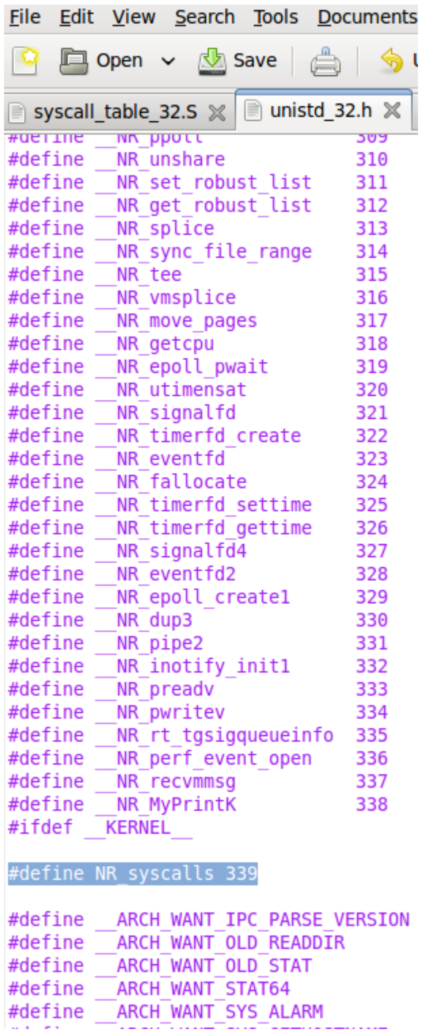
System call registration III



We also register our system call here (the number we write on the right is our syscall's number)

Our syscall's number

System call registration IV

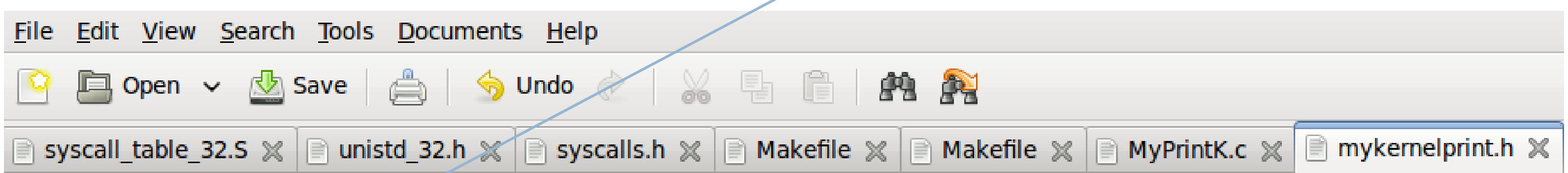


We also need to update this number to (last syscall number) +1

The userspace .h file

Here we define the userspace envelope for our system call

Note that we define our system call here




```c
#include<linux/unistd.h>
#define __NR_MyPrintK 338

long MyPrintK(const char* myStr)
{
        return syscall(__NR_MyPrintK, myStr);
}
```

The userspace driver

We include our userspace envelope
in our driver file