# Assignment 5:
## Adding and testing a new system call to Linux kernel

Antonis Papadogiannakis

HY345 – Operating Systems Course

# Outline

- Introduction: system call and Linux kernel
- Emulators and Virtual Machines (demo with QEMU)
- Compile Linux kernel 2.6 (demo with linux-2.6.38.1)
- Load a new kernel with QEMU (demo)
- Basic steps to add a new system call to Linux kernel (example)
- How to use the new system call (example)
- The new system call in this assingment: *getproctimes*
- Several hints

# System call

- System call: an interface between a user-level program and a **service** provided by kernel
  - Implemented in kernel
  - With a user-level interface
  - Crossing the user-space/kernel-space boundaries

- Trap: switch to kernel mode
  - e.g. when calling a system call

# Linux kernel

- Popular
- Open source
- www.kernel.org

- Extending the Linux kernel
  - Usually with loadable kernel modules
- Architecture: monolithic kernel
  - A set of **system calls** implement all Operating System services
  - User space, kernel space boundaries
- Preemptive scheduling, virtual memory

# Emulators

- Enable us to emulate an Operating System (**guest OS**) using another Operating System (**host OS**)
  - e.g. running Windows from a Linux OS
  - or running multiple OS in a single computer
  - as a simple user, in user-level
  - Guess OS can crash without affecting host OS
  - thus very useful for **kernel development** and **debugging**

# The QEMU emulator

- Fast open source emulator
- You will use it in this assignment
- www.qemu.org
- Installed in CSD machines

```
$ qemu –hda disk.img
```

- Virtual disk image (disk.img)
  - Like a common disk
  - We can install an OS distribution into this image
  - hy345-linux.img is the disk image you will use in this assingment, with a minimal Linux installation and kernel 2.6.38.1
- Host OS: a CSD machine   Quest OS: ttylinux

# Demo

with QEMU

# Linux kernel 2.6.38.1

- Get the code from *~hy345/qemu-linux/linux-2.6.38.1.tar.bz2*
  - or from www.kernel.org
- View source
  - Organized in **kernel**, mm, drivers, etc
  - We are mostly interested in files in **kernel** folder
  - Headers are in the **include** folder
  - x86 32-bit architecture (i386)
  - Use **grep**, find, ctags

# Compile the Linux kernel

- 2 steps
  - Configure
    - `make config`, `make menuconfig`, etc
    - produce .config file
    - We give you directly the proper .config file, so no need for configuring kernel
  - Build
    - `$ make ARCH=i386 bzImage`
    - Builds Linux kernel image for i386 architecture
    - *linux-2.6.38.1/arch/x86/boot/bzImage*
    - bzImage used to boot with QEMU with the new kernel
  - (we do not consider install in this assignment due to emulator-based testing)

# Demo

with linux-2.6.38.1 kernel source code and compilation

# Load the new kernel with QEMU

```
$ qemu –hda hy345-linux.img –append “root=/dev/hda”
  –kernel linux-2.6.38.1/arch/x86/boot/bzImage
```

- Use the same disk image hy345-linux.img as /dev/hda
- This image contains the root filesystem
- Load OS with the new kernel image

```
$ uname –a
```

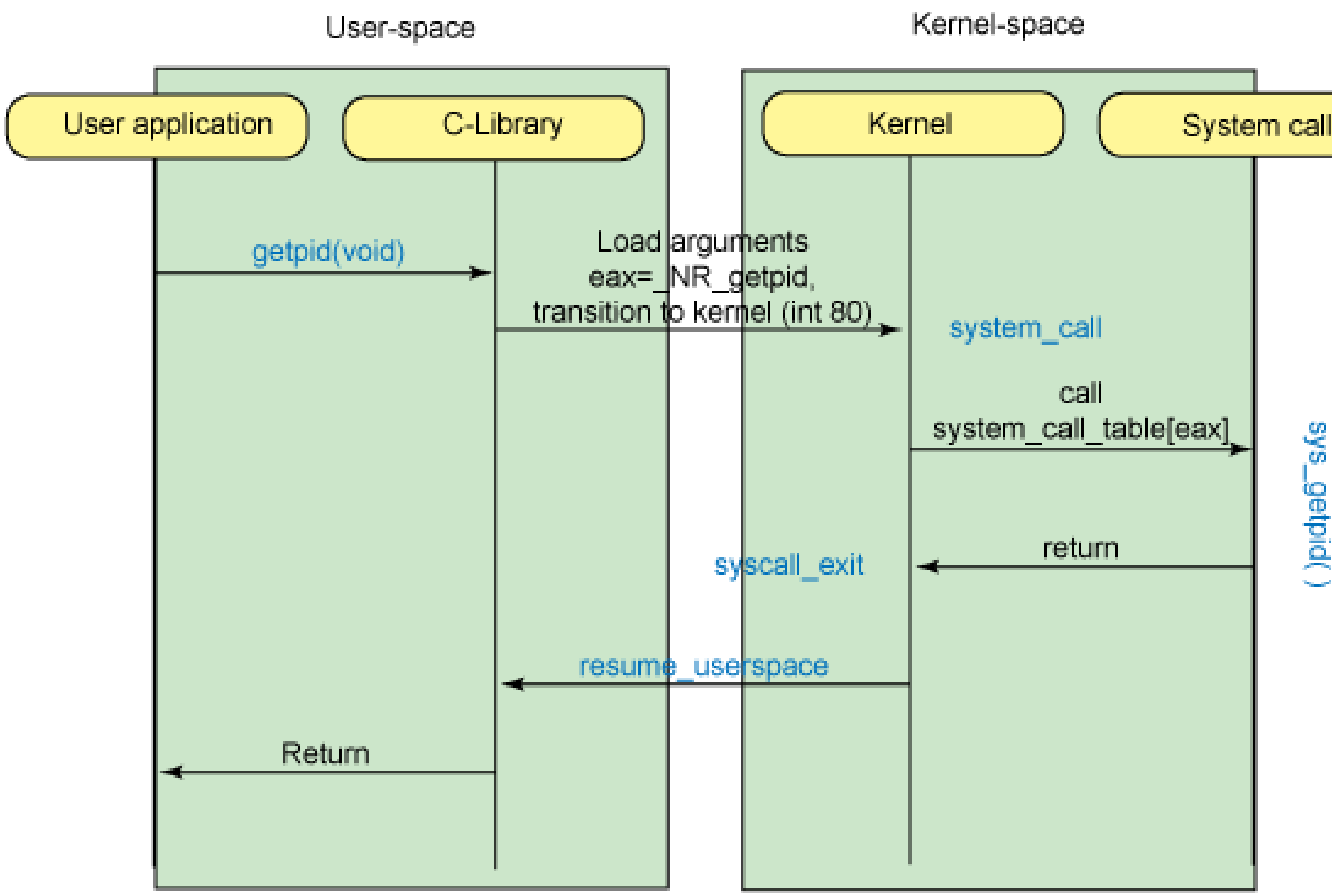
  - To find the kernel version
  - Append your **username** in the kernel version and use **revision** numbers for your convenience

- Compile Linux kernel in host OS, boot with the new kernel in guest OS

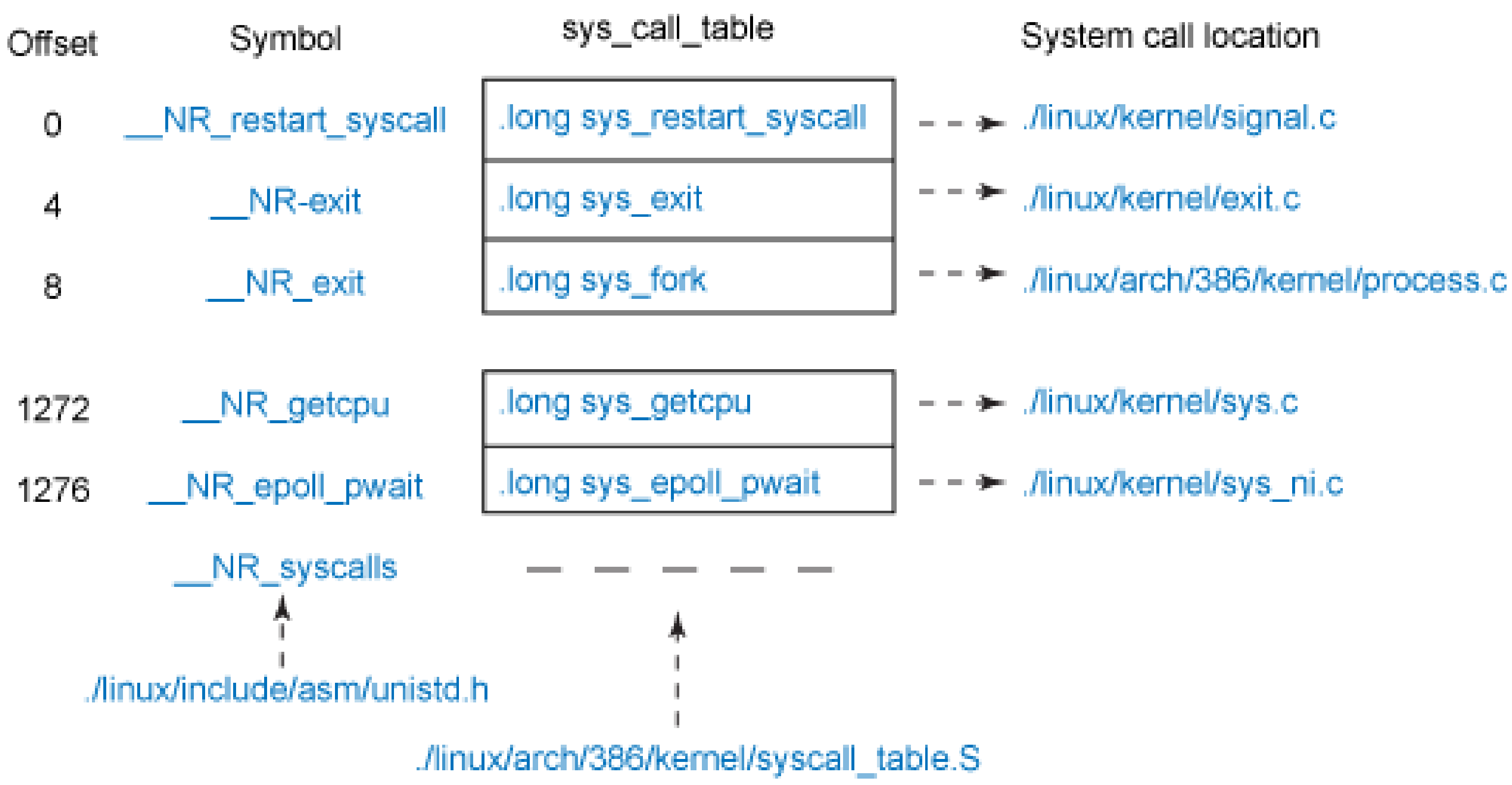
# Demo

Loading QEMU with a new kernel image built in the host OS

# Control flow of a system call in Linux kernel



# System call table



# Three basic steps to add a new system call in Linux kernel

1. Add a new system call number N
2. Add a new system call table entry for the above system call number N with a function pointer to function F
3. Implement  the function F with system call's actual functionality.
   - Also add proper header files for new types
   - Copy arguments from user space to kernel and results from kernel to user space

# An example: *dummy_sys*

- The *dummy_sys* system call takes one integer as single argument
- It prints this argument in kernel and returns this integer multiplied by two

# Step 1: Add new system call number

- *Open linux-2.6.38.1/arch/x86/include/asm/unistd_32.h*
- Find system call numbers
- Find last system call number (340)
- Define a new one with the next number (341)

```
#define __NR_dummy_sys 341
```

- Increase NR_syscalls by one (341 -> 342)

- *dummy_sys* has the **341** system call number

# Step 2: Add new entry to system call table

- Open *linux-2.6.38.1/arch/x86/kernel/syscall_table_32.S*
- Add in the last line the name of the function that implements the *dummy_sys* system call

```
.long sys_dummy_sys /* 341 */
```

- *sys_dummy_sys* function will implement the *dummy_sys* system call

# Step 3: Implement the system call's function

- Create *linux-source-2.6.38.1/kernel/dummy_sys.c*
- Write system call's functionality

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

asmlinkage long sys_dummy_sys(int arg0)
{
   printk("Called dummy_sys with argument: %d\n",arg0);
   return((long)arg0*2);
}
```

# Arguments by reference

- Strings, pointers to structures, etc

```
int access_ok( type, address, size );

unsigned long copy_from_user( void *to, const void __user
    *from, unsigned long n );

unsigned long copy_to_user( void *to, const void __user *from,
    unsigned long n );
```

# Using the new system call

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#define __NR_dummy_sys 341

int main() {
   printf("Trap to kernel level\n");
   syscall(__NR_dummy_sys, 42);
   //you should check return value for errors
   printf("Back to user level\n");
}
```

# Wrapper function

- Define a macro

```
#define dummy_sys(arg1) syscall(341, arg1)
```

- Write a wrapper function

```
long dummy_sys(int arg1) {
  syscall(341, arg1);
}
```

- So in the test program we just call

```
dummy_sys(42);
```

# The *getproctimes* system call

```
int getproctimes(int pid, struct proctimes *pt);
```

- First argument: pid of a process, or the current process if (pid==-1)
- Second argument: passed by referenced and used by kernel to return the necessary info to user space
- Return value: EINVAL on error or 0 on success

# The *struct proctimes*

- Should be defined in a new file:

*linux-2.6.38.1/include/proctimes.h*

```
struct proctimes { //info and times about processes we need
   struct proc_time proc; //process with pid or current process
   struct proc_time parent_proc; //parent process
   struct proc_time oldest_child_proc; //oldest child process
   struct proc_time oldest_sibling_proc; //oldest sibling process
}
```

# The *struct proc_time*

- Also defined in *linux-2.6.38.1/include/proctimes.h*

```
struct proc_time { //info and times about a single process
    pid_t pid; //pid of the process
    char name[16]; //file name of the program executed
    unsigned long start_time; //start time of the process
    unsigned long real_time; //real time of the process execution
    unsigned long user_time; //user time of the process
    unsigned long sys_time; //system time of the process
}
```

# In every execution of *getproctimes*

- Every time the getproctimes is executed in kernel you should print a message
  - Using **printk**
  - The message will include your full name and A.M.
  - You can view these messages from user level upon the execution of *getproctimes* with “*dmesg”* or “*cat /var/log/messages*”

- **printk** is very useful for debugging messages

# Testing *getproctimes*

- You should write several test programs in the guest OS using *getproctimes*
  - To validate its correct operation
- We require three test programs
  1. Get the times of the current process with *getproctimes* when it performs 1M multiplications and **sleep**(5)
  2. Compare info and times of all processes when calling multiple **fork**()
  3. Get *pid* from command line and call *getproctimes* with this *pid*, and use *pids* from **ps**
- Any other test program you think useful

# Hints

- To calculate real time you shoud read current time
  - see *linux-2.6.38.1/include/linux/time.h* and *gettimeofday* system call
- To find the current process
  - see *linux-2.6.38.1/include/asm/current.h*
- To find info and times about each process
  - see *task_struct* in *linux-2.6.38.1/include/linux/sched.h*
- In *tast_struct* you can also find
  - the parent process
  - child processes list (and find the oldest)
  - sibling processes list (and find the oldest)
- See existing system calls: *getpid*, *gettimeofday*, *times*

# What to submit

1. bzImage
2. Only modified or created by you source files of the Linux kernel 2.6.38.1 (both C and header files)
3. Test programs and header files used in the guest OS for testing the new system call
4. README with implementation details and experiences from testing

Gather these files in a single directory and send them using the submit program as usual

# Good luck

Deadline: 14/12