

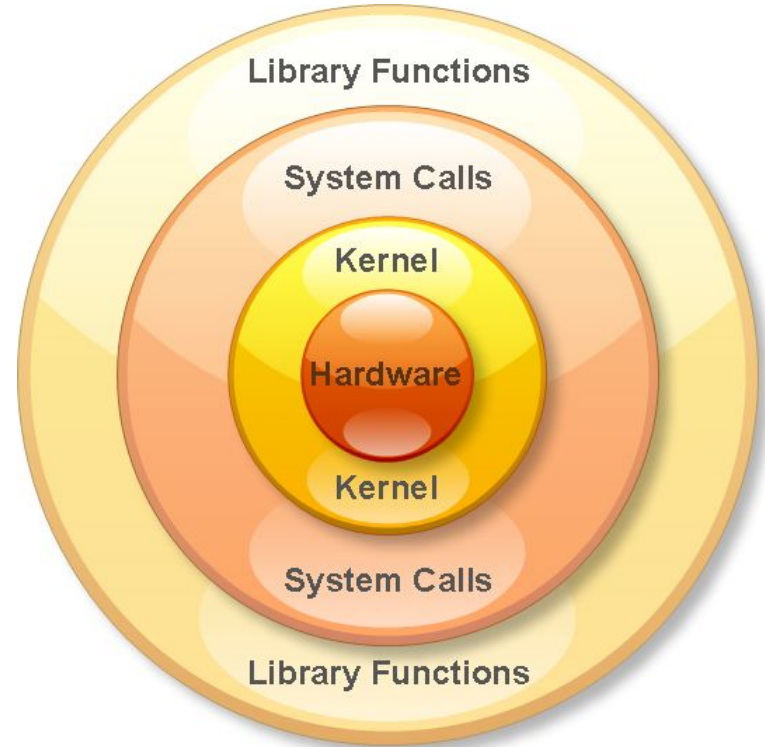
System Calls

(Φροντιστήριο για την 3η σειρά)

Dimitris Deyannis
deyannis@csd.uoc.gr

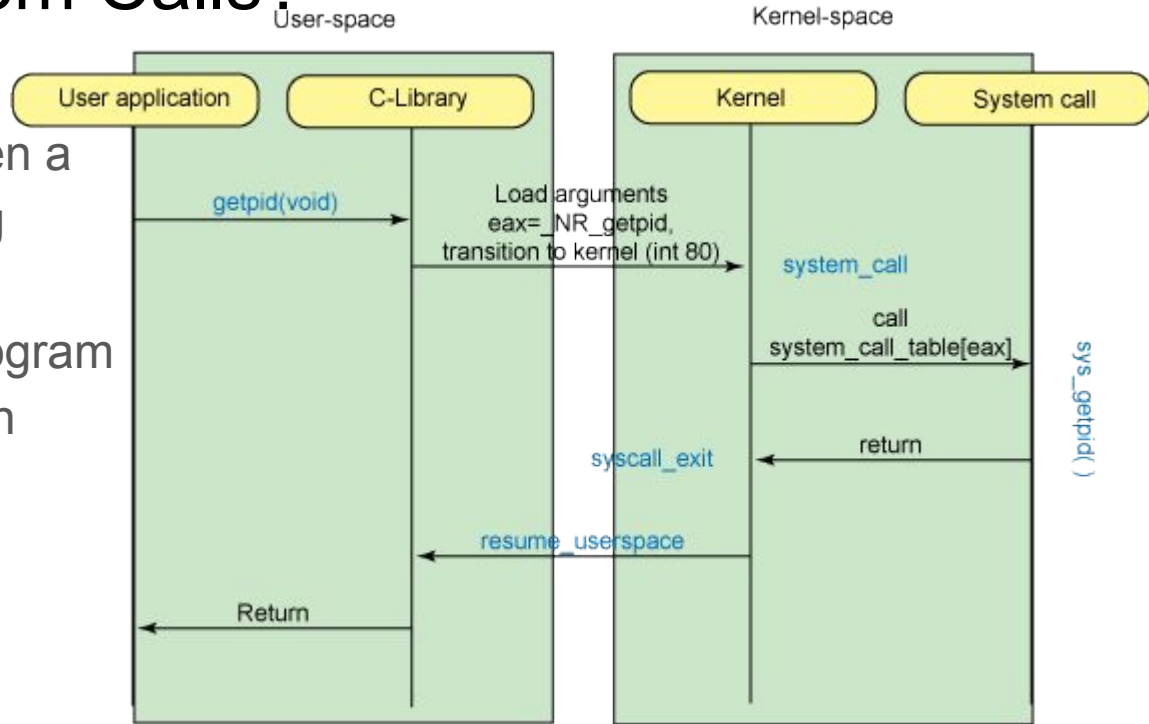
What is a System Call?

- The system call is the fundamental interface between an application and the Linux kernel



Why we need System Calls?

- System calls provide an essential interface between a process and the operating system
- A system call is how a program requests a service from an operating system's kernel



What can System Calls do?

- File management
 - create, open, delete..
- Process control
 - exec, kill, wait...
- Device management
 - request, release...
- Information maintenance
 - get time, set time...
- Communication
 - sockets, send, receive...

How do we use System Calls?

- `sys/syscall.h` is a small library that implements `long syscall(long number, ...);`
- This function invokes the system call that corresponds to the “number” while “...” corresponds to the rest of the arguments

Preparation

```
$ mkdir /spare/[username]
```

```
$ chmod 700 /spare/[username]
```

```
$ cp ~hy345/qemu-linux/hy345-linux.img /spare/[username]/
```

```
$ cp ~hy345/qemu-linux/linux-2.6.38.1.tar.bz2 /spare/[username]/
```

```
$ cd /spare/[username]
```

```
$ tar -jxvf linux-2.6.38.1.tar.bz2
```

Edit kernel source code to implement the new system calls

Implementing a new System Call

1. Define a system call number
2. Define a function pointer
3. Define a function
4. Implement the system call

Define a System Call number

- Every system call has an invocation number
- Edit: `linux-2.6.38.1/arch/x86/include/asm/unistd_32.h`
 - Define the new system call number at the bottom of the list
 - e.g. `#define __NR_dummy_sys 341`
 - Update the number of system calls
 - `#define NR_syscalls 342`

Define a function pointer

- The Kernel needs to have a function pointer pointing to the new system call
- Edit: `linux-2.6.38.1/arch/x86/kernel/syscall_table_32.S`
- Define the function pointer at the bottom of the list
 - e.g. `.long sys_dummy_sys /* 341 */`

Define a function

- We have to define the function signature in syscalls.h file
- Edit: linux-2.6.38.1/include/asm-generic/syscalls.h
- At the bottom of the file add:

```
#ifndef dummy_sys
    asmlinkage long sys_dummy_sys(int arg0);
#endif
```

Implement the System Call part 1

- Touch and edit: linux-2.6.38.1/kernel/dummy_sys.c as such:

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

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

Implement the System Call part 2

- Edit: `linux-2.6.38.1/kernel/Makefile`
- Add: `obj-y += dummy_sys.o`

- Now you are ready to compile the Kernel with your new system call!

Compile the Linux Kernel

Edit kernel source code to implement the new system calls

```
$ cp ~hy345/qemu-linux/.config . << Mind the dot!!
```

Edit .config, find CONFIG_LOCALVERSION="-hy345", and append to the kernel's version name your username and a revision number

```
$ make ARCH=i386 bzImage
```

Create a simple test

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>

#define __NR_dummy_sys 341

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

Test the system call

- Start the VM using the new kernel
 - `$ qemu-system-i386 -hda hy345-linux.img -append "root=/dev/hda" -kernel linux-2.6.38.1/arch/x86/boot/bzImage -curses`
- Transfer the test file into the VM
 - `$ scp [username]@10.0.2.2:/path/to/thest/test.c . << Mind the dod!!`
- Compile the test
 - `$ gcc -o test test.c`
- Run the test
 - `$./test`
- Check the kernel log
 - `$ dmesg | tail`

What a process does

- A process declares the time it needs to be executed
- If a process does not declare the time, we assume its infinite
- The process should be able to set **ONLY** its own parameters or the ones of its child processes

What the kernel does

- At each scheduling interval the kernel chooses the process with the "Least Remaining Time"
- Then it runs it for a quantum and updates the necessary values
- If the "Least Remaining Time" for a process starts to become negative, the scheduler penalizes the process by turning it into infinite

Implementation

- For this assignment you have to implement the following system calls
 - `set_total_computation_time(int pid, unsigned int total_time)`
 - `get_remaining_time(int pid, struct t_params *t_arguments)`

Implementation

- Add 3 new fields in `task_struct`
 - `unsigned int total_computation_time;` // The time requested
 - `int remaining_time;` // The remaining time
 - `unsigned int infinite;` // A flag to declare infinite
- Implement the `t_params` struct
 - `int remaining_time;`
 - `unsigned int infinite;`