# Operating systems

## System Call

## Lecture 7 Part 2

Florenc Demrozi

florenc.demrozi@univr.it

University of Verona
Department of Computer Science

2021/2022

# Table of Contents



# Preparation

# Preparation

## Switch to Exercise branch

1. Save your work!!!
   - *e.g.*, **MentOS/src/process/scheduler_algorithm.c**
2. `git reset --hard`
3. `git pull`
4. `git checkout --track origin/feature/Feature-DeadlockExercise`

# System Call

# System Call

## What are the ingredients in MentOS

# System Call

## Ingredients

The ingredients are:

- 1 **kernel-side** function;
- 1 **user-side** function;
- 1 **unique number** associated with the system call;

For instance:

- **kernel-side** function:

```
int sys_open(const char *pathname, int flags, mode_t mode);
```

- **user-side** function:

```
int open(const char *pathname, int flags, mode_t mode);
```

- **unique number** associated with the system call:

```
#define __NR_open 5
```

# System Call

## Folder Structure

**inc/sys/unistd.h**

- The file **defining** the **user-side** system calls;
- For instance, it contains the **open(...)** function.

**src/libc/unistd/*.c**:

- The files **implementing** the **user-side** system calls;
- Basically, they prepare the **arguments**, and call **int 80**.
- The **open(...)**, is implemented inside **src/libc/unistd/open.c**

**inc/system/syscall_types.h**

- Contains the list of **System Calls numbers**;
- The `#define __NR_open 5`;

# System Call

## How they work in MentOS

# System Call

## How they work in MentOS

```
fd = open(filename, flags, mode);
```

main.c

**open(...)**

```
mov eax __NR_open
mov ebx filename
mov ecx flags
mov edx mode
int $0x80
```

src/libc/unistd/open.c

**syscall_handler(...)**

```
// The System Call number is in EAX.
sc_ptr = sc_table[regs->eax];

// Call the SC, the arguments are in EBX, EXC and EDX.
regs->eax = sc_ptr(regs->ebx, regs->ecx, regs->edx);
```

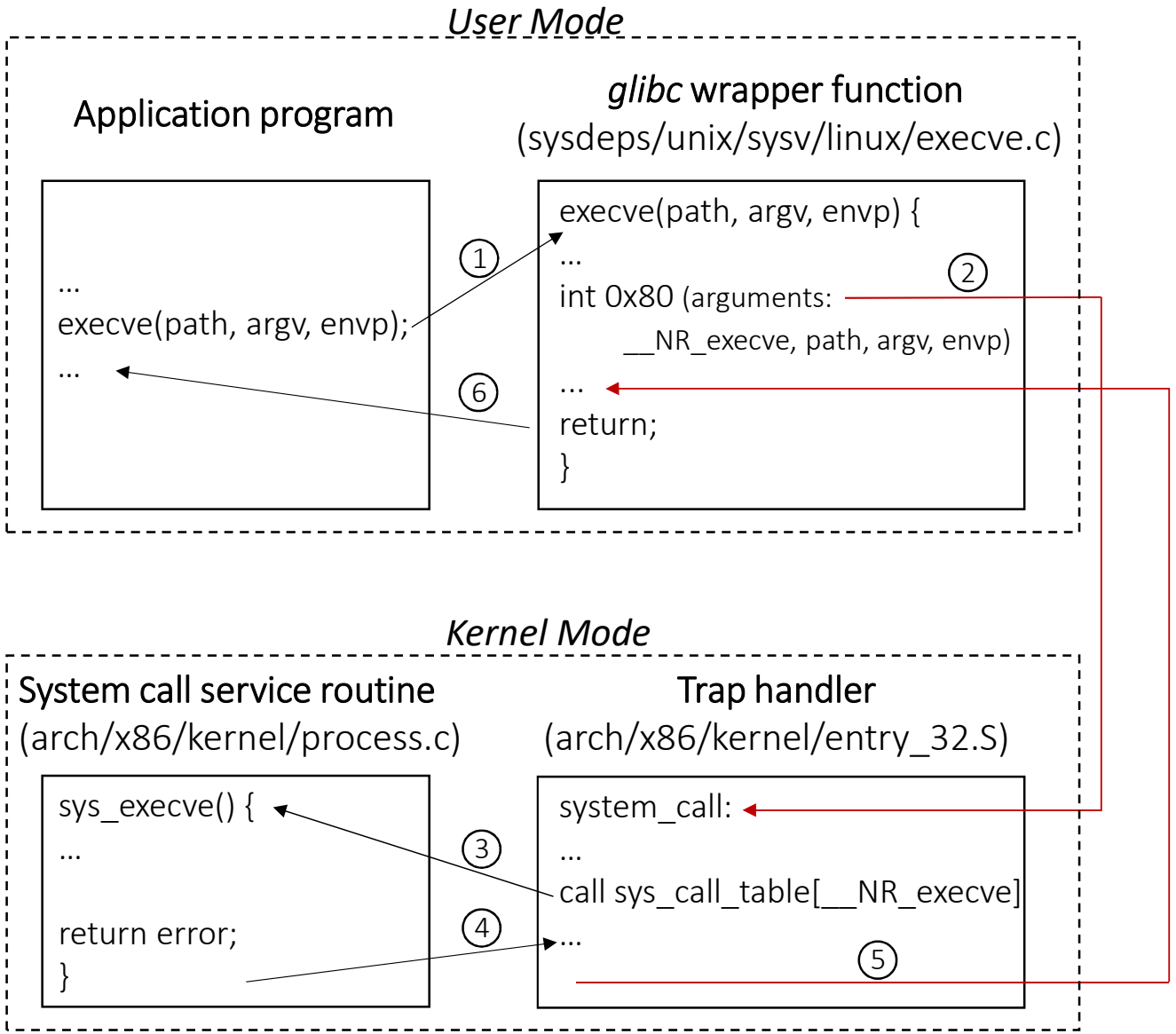
src/system/syscall.c

**sys_open(...)**

```
// Open the file with filesystem.
```

src/fs/open.c

# System call execution




**Preparing the registers** is done through easy-to-use **macros**:

```
int open(const char *pathname, int flags, mode_t mode) {
  ssize_t retval;
  DEFN_SYSCALL3(retval, __NR_open, pathname, flags, mode);
  if (retval < 0)
    errno = -retval, retval = -1;
  return retval;
}
```

```
int close(int fd) {
  int retval;
  DEFN_SYSCALL1(retval, __NR_close, fd);
  if (retval < 0)
    errno = -retval, retval = -1;
  return retval;
}
```

# System Call

## How can I add a new one in MentOS

# System Call

## Assumption 1

Let us assume we **already have** the following **kernel-side** functions

**inc/experimental/smart_sem_kernel.h**,
**src/experimental/smart_sem_kernel.c**:

- `int sys_sem_create();`
  Smart semaphore creation.
- `int sys_sem_destroy(int id);`
  Destruction of a created smart semaphore.
- `int sys_sem_init(int id);`
  Initialization of a created smart semaphore.
- `int sys_sem_try_acquire(int id);`
  Tries a safety acquisition of a smart semaphore identified by an ID and, if available, takes the ownership.
- `int sys_sem_release(int id);`
  Release the ownership of a smart semaphore.

# System Call

## Assumption 2

Let us also assume we **already have** the following **user-side** functions

**inc/experimental/smart_sem_user.h**,
**src/experimental/smart_sem_user.c**:

- `int sem_create();`
- `int sem_destroy(int id);`
- `int sem_init(int id);`
- `int sem_acquire(int id);`
- `int sem_release(int id);`

But, **empty**!

# System Call

## How we add these new system call?

In order to add these new System Calls, you have to:

1. Go inside **inc/system/syscall_types.h** and chose a name for your system call number:
   ```
   #define __NR_sem_...  190
   ```
2. Go inside **src/system/syscall.c** and register/associate your system calls and their numbers inside the **system call table**:
   ```
   sys_call_table[__NR_sem_...]  = (SystemCall)sys_sem_...;
   ```
3. Go inside **src/experimental/smart_sem_user.c**, and fill the user-side system calls, and (by taking as inspiration the other user-side system calls) write the actual call through the MACROS!

**Beware**: the **acquire** function user-side is called **sem_acquire**, while on the kernel-side we have the **sem_try_acquire**. Just, be careful with the names.