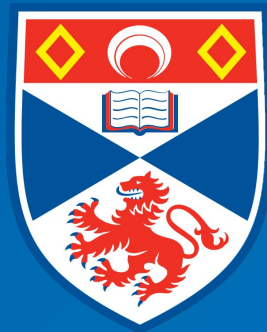


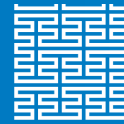
Propagating C++ Exceptions across the User/Kernel Boundary

Dmitry Voronetskiy and **Tom Spink**
University of St Andrews





A little bit about me



Tom Spink

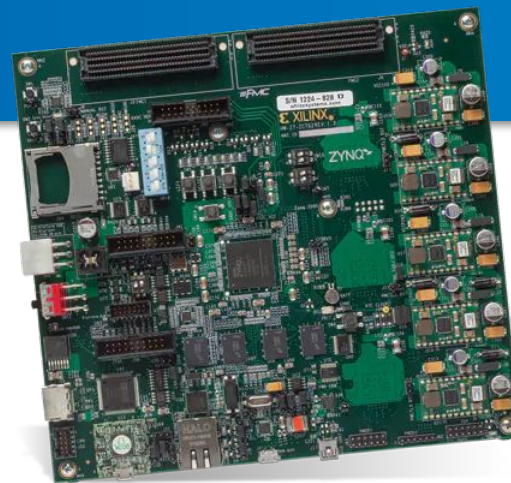
Lecturer, School of Computer Science
FBCS

Interested in:

- Dynamic Binary Translation
- Virtualisation
- Hardware acceleration
- Operating Systems
- Compilers
- Trains

And more generally...

- Computer systems!



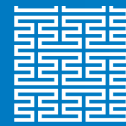
Programming Languages

Operating Systems

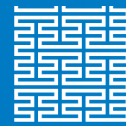
Error codes



EBADF



EFAULT



EINVAL



No error handling



```
int fd = open("some file", O_RDONLY);  
read(fd, buffer, 128);  
close(fd);
```



Some error handling



```
int fd = open("some file", O_RDONLY);  
  
if (fd < 0) {  
    printf("an error occurred\n");  
    return 1;  
}  
  
read(fd, buffer, 128);  
  
close(fd);
```



Better error handling



```
int fd = open("some file", O_RDONLY);  
  
if (fd < 0) {  
    perror("error opening file");  
    return 1;  
}  
  
read(fd, buffer, 128);  
  
close(fd);
```



But what is the error?



open(2)

System Calls Manual

open(2)

NAME top

open, openat, creat - open and possibly create a file

LIBRARY top

Standard C library (libc, -lc)

SYNOPSIS top



But what is the error?



ERRORS

top

`open()`, `openat()`, and `creat()` can fail with the following errors:

EACCESS The requested access to the file is not allowed, or search permission is denied for one of the directories in the path prefix of `pathname`, or the file did not exist yet and write access to the parent directory is not allowed. (See also `path_resolution(7)`.)

EACCESS Where `O_CREAT` is specified, the `protected_fifos` or `protected_regular` `sysctl` is enabled, the file already exists and is a FIFO or regular file, the owner of the file is neither the current user nor the owner of the containing directory, and the containing directory is both world- or group-writable and sticky. For details, see the descriptions of `/proc/sys/fs/protected_fifos` and `/proc/sys/fs/protected_regular` in `proc_sys_fs(5)`.

EBADF (`openat()`) `pathname` is relative but `dirfd` is neither `AT_FDCWD` nor a valid file descriptor.

EBUSY `O_EXCL` was specified in `flags` and `pathname` refers to a block device that is in use by the system (e.g., it is mounted).

EDQUOT Where `O_CREAT` is specified, the file does not exist, and the user's quota of disk blocks or inodes on the filesystem has been exhausted.

EEXIST `pathname` already exists and `O_CREAT` and `O_EXCL` were used.

EFAULT `pathname` points outside your accessible address space.

EFBIG See `Eoverflow`.

EINTR While blocked waiting to complete an open of a slow device (e.g., a FIFO; see `fifo(7)`), the call was interrupted by a signal handler; see `signal(7)`.



But what is the error?



EINVAL The filesystem does not support the O_DIRECT flag. See NOTES for more information.

EINVAL Invalid value in flags.

EINVAL O_TMPFILE was specified in flags, but neither O_WRONLY nor O_RDWR was specified.

EINVAL O_CREAT was specified in flags and the final component ("basename") of the new file's pathname is invalid (e.g., it contains characters not permitted by the underlying filesystem).

EINVAL The final component ("basename") of pathname is invalid (e.g., it contains characters not permitted by the underlying filesystem).



A **rich exception object** can contain a lot more context and detail about an error.



Exception-based Handling



```
int fd;
try {
    fd = open("some file", O_RDONLY);
} catch (const invalid_flags_exception& ex) {
    printf("invalid flag in open: %x\n", ex.invalid_flags);
    return 1;
}

read(fd, buffer, 128);
close(fd);
```



Exception Generation in the Standard Library



```
int file::open(const string& pathname, open_flags flags) {  
    int fd = open(pathname, flags);  
    if (fd < 0) {  
        errno_to_exception();  
    }  
  
    return fd;  
}  
  
void errno_to_exception()  
{  
    switch (errno) {  
        case EINVAL:  
            throw invalid_value_exception();  
        default:  
            throw unknown_error_exception();  
    }  
}
```

What if we raised the exception in the kernel, and it propagated through to userspace?



Exception-based Handling



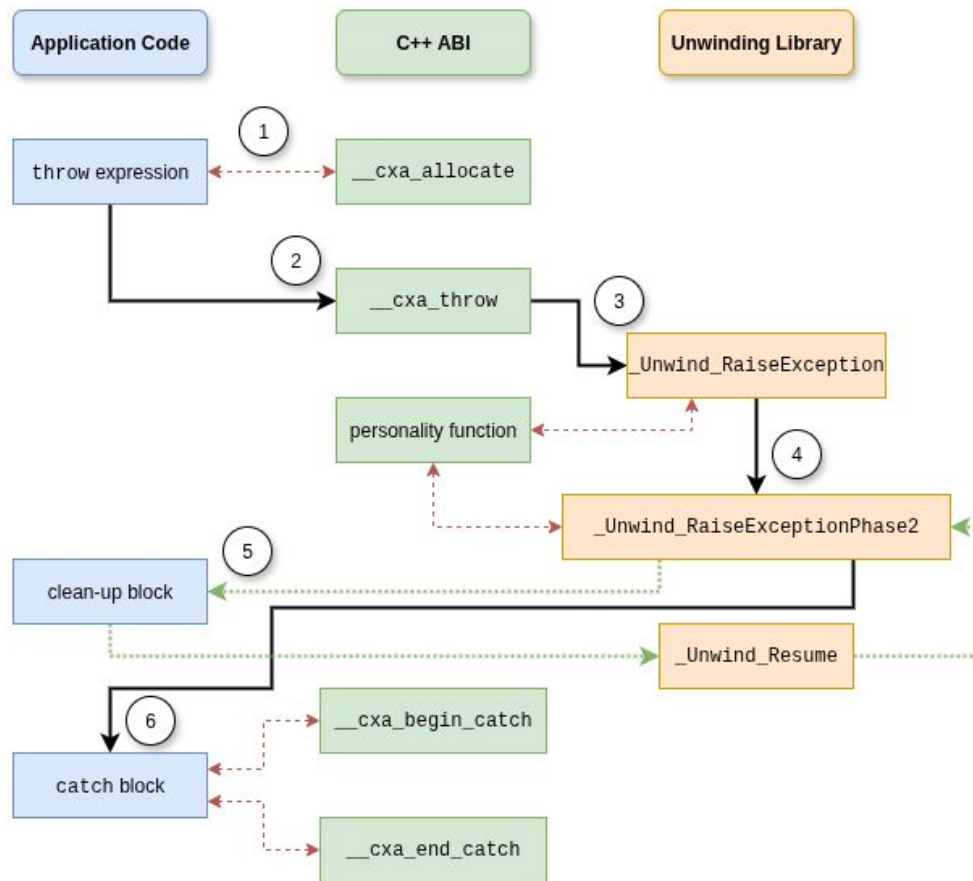
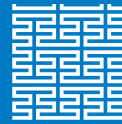
```
int fd;
try {
    fd = open("some file", O_RDONLY);
} catch (const invalid_flags_exception& ex) {
    printf("invalid flag in open: %x\n", ex.invalid_flags);
    return 1;
}

read(fd, buffer, 128);
close(fd);
```

C++ Exception Mechanism/ABI



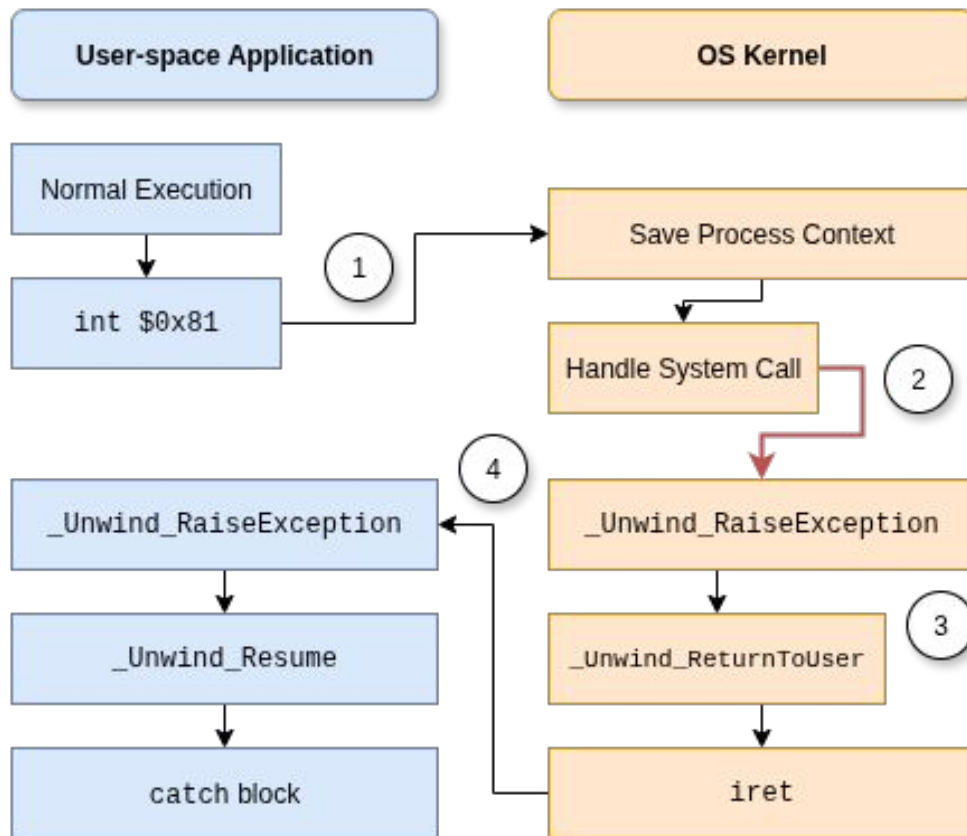
Exceptions in C++



Crossing the boundary

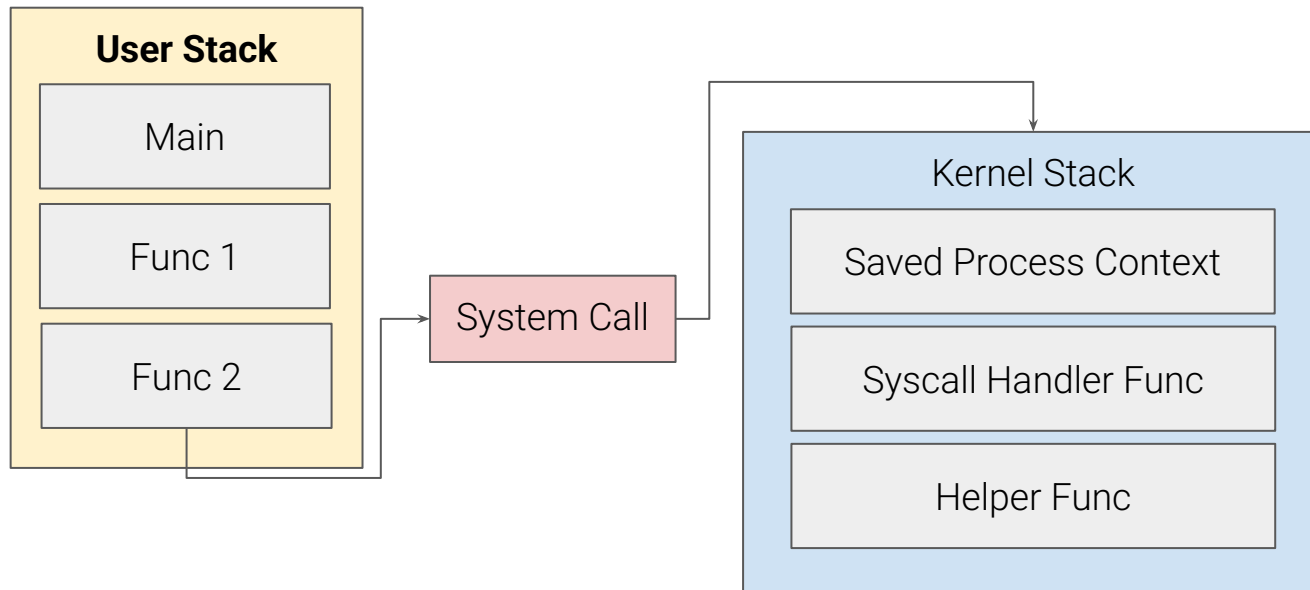


System Call Throwing



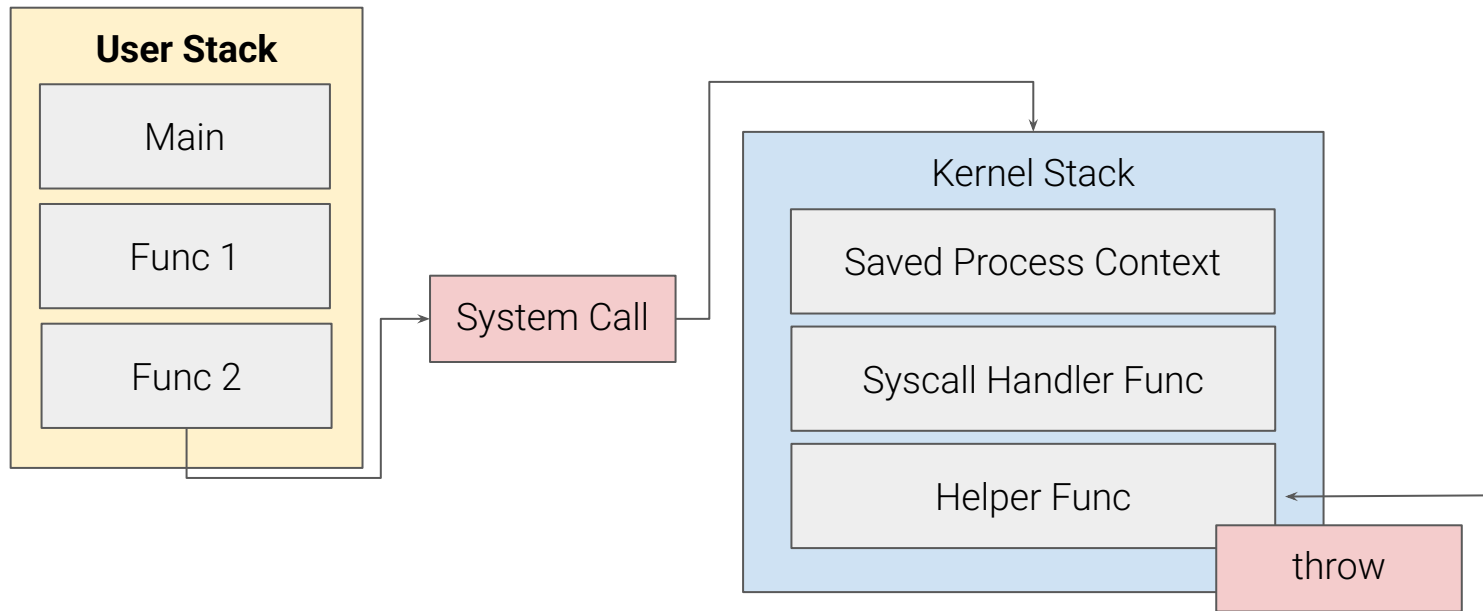


System Call Stack



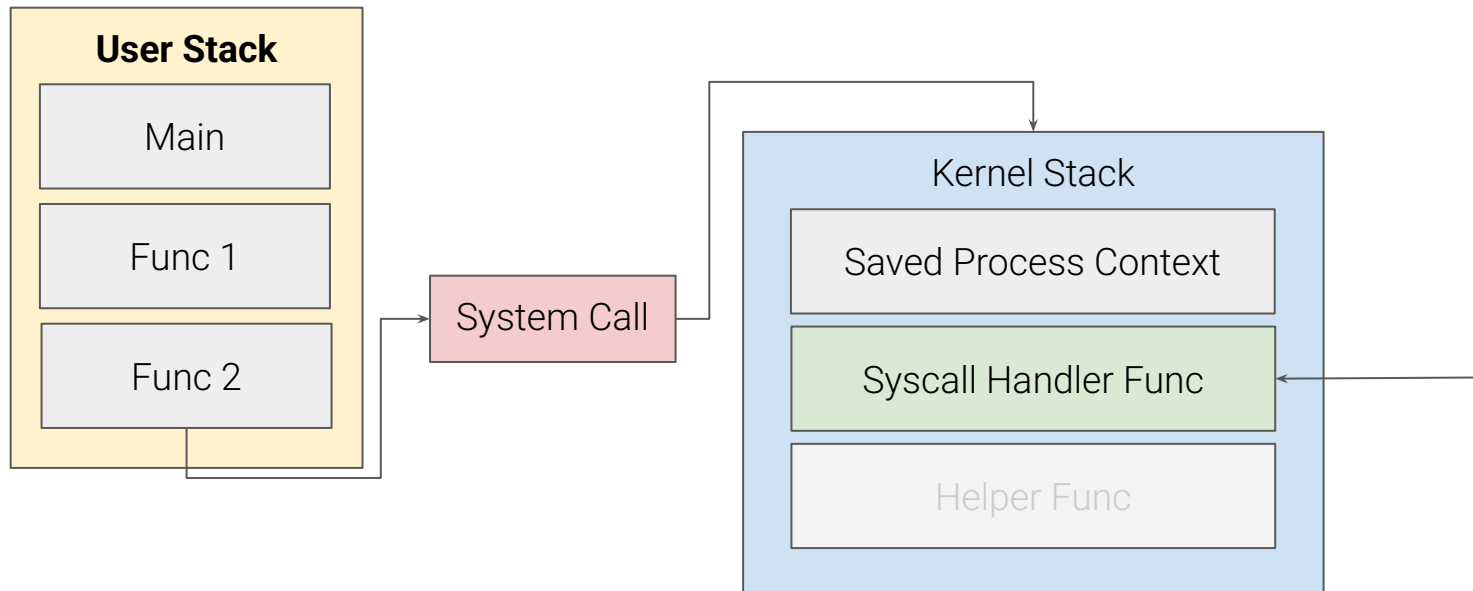


System Call Stack



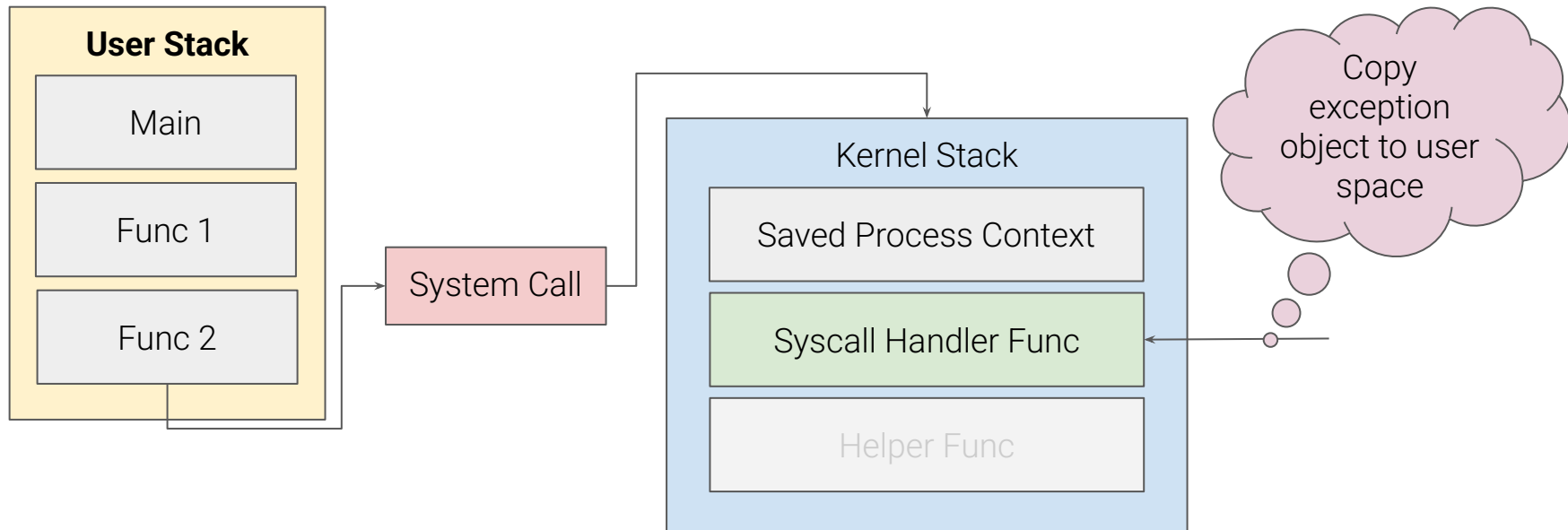


System Call Stack



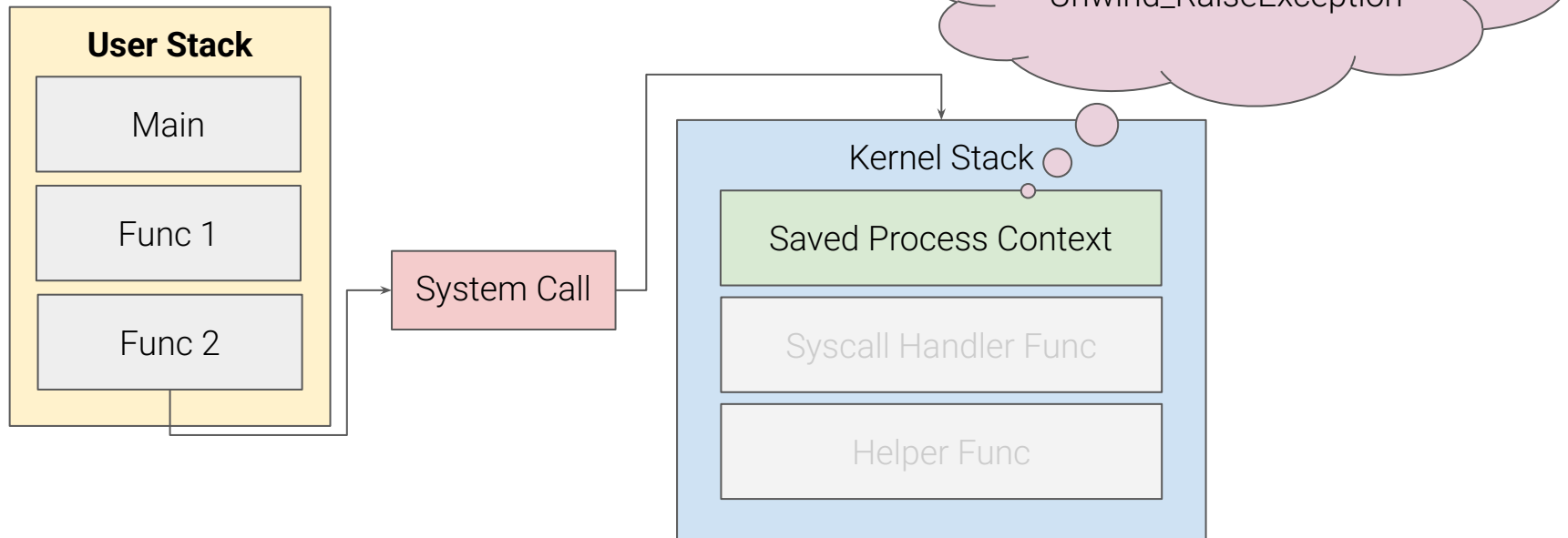


System Call Stack



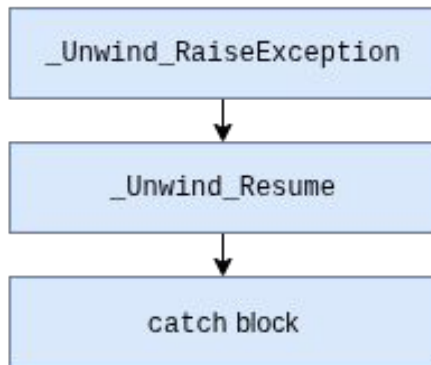
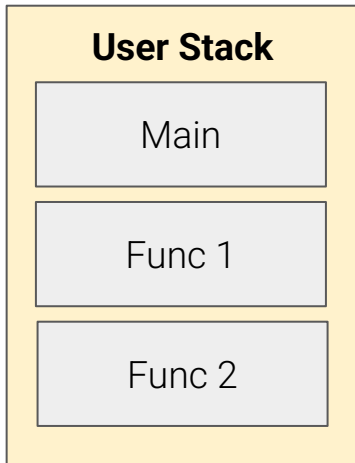


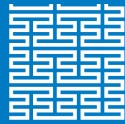
System Call Stack





System Call Stack





The exception object **allocated** in kernel space would be **inaccessible** to user code, so it must be **copied**.

This is the trickiest part, and is what places most limitations on the technique.



When a user thread starts, it tells the kernel where its **exception storage buffer is, and notifies it where the “**_Unwind_RaiseException**” function is.**

During exception propagation, the kernel locates the buffer for the appropriate thread, and copies the object in.

**What about hardware
exceptions?**



Exception Generation in the Standard Library



```
long random() {  
    try {  
        long result ;  
        asm("rdrand %0" : "=r"(result));  
        return result;  
    } catch (const illegal_instruction_exc& e) {  
        return generate_random_slow();  
    }  
}
```

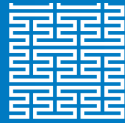


Exception Generation in the Standard Library



```
long random() {  
    try {  
        long result ;  
        asm('rdrand %0' : "=r"(result));  
        return result;  
    } catch (const illegal_instruction_exc& e) {  
        return generate_random_slow();  
    }  
}
```

```
.globl illegal_instruction_handler  
illegal_instruction_handler:  
    <save context>  
    call handle_illegal_instruction  
    <restore context>  
    iret
```



Page faults can be kept in the **context** of the code causing them, rather than a “**global**” signal handler.

```
int read_state(object *obj) {  
    try {  
        return obj->state;  
    } catch (const null_pointer_exception& e) {  
        throw invalid_object_ptr();  
    }  
}
```

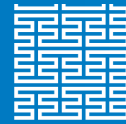
Analysis



- Exception handling is **slow**.
 - Throwing and catching an exception takes 18x longer, than doing nothing.
- But, in the non-exceptional case, it's **faster**; as there are fewer tests on the “hot path”
- Binaries get **bigger**.
- Application ecosystem needs to **support exceptions**.



Thanks!



Any questions?

Tom Spink

University of St Andrews

tcs6@st-andrews.ac.uk



Error Propagation



```
fn might_error(val: i32) -> Result<i32, i32>
{
    if val < 0 {
        Ok(val * 2)
    } else {
        Err(0)
    }
}
```

```
fn test(val: i32) -> Result<i32, i32> {
    let result = might_error(val)?;
    Ok(result);
}
```



Error Propagation



```
pub enum Flags
{
    None,
    Direct
}

pub enum SomeKindOfError
{
    BadFlags(Flags)
}

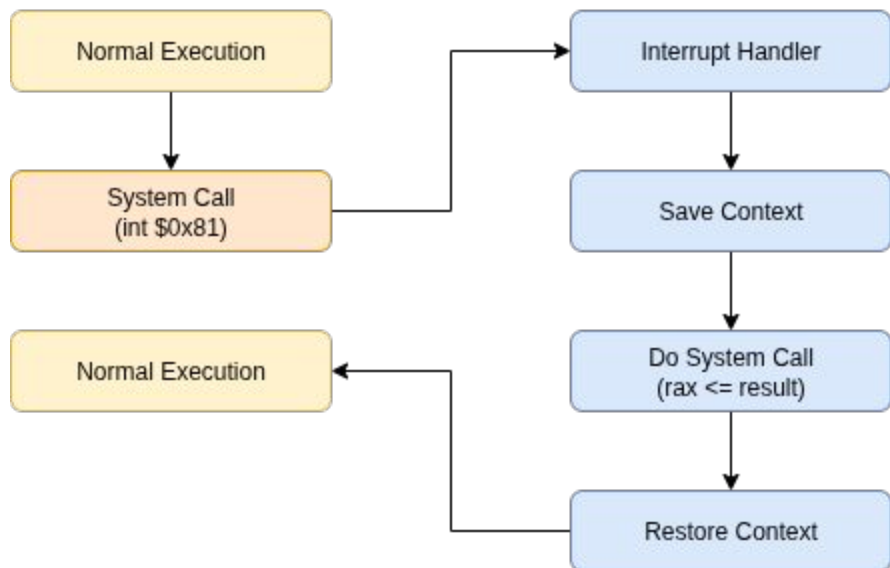
pub fn open(path: PathBuf, flags: Flags) -> Result<i32, SomeKindOfError>
{
    if flags == Flags::Direct {
        Err(SomeKindOfError::BadFlags(Flags::Direct))
    } else {
        Ok(1)
    }
}
```



- InfOS (no longer maintained):
<https://github.com/tspink/infos>
- StACSOS (actively maintained!):
<https://github.com/tspink/stacsos>

User-space

Kernel-space



User-space

Kernel-space

