Lecture 4 — Interrupts & System Calls

Jeff Zarnett & Mike Cooper-Stachowsky
jzarnett@uwaterloo.ca, mstachowsky@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

March 10, 2024

OBJECTION!!!

Interrupts

The CPU needs data, but it takes a variable amount of time to get it.
    Sometimes this means the equivalent of walking a book from Ottawa.

In the meantime, I should do something else.

Polling: check periodically if the book has arrived.

Interrupts: get a notification when the book is here.

If someone knocks on my door, I pause what I'm doing and get the book.

Sources of Interrupts

We can put interrupts into four categories, based on their origin:

1. **Program.**
2. **Timer.**
3. **Input/Output.**
4. **Hardware Failure.**

Interrupts

Interrupts are a way to improve processor utilization.

CPU time is valuable!

When an interrupt take place, the CPU might ignore it (rarely).

More commonly: we need to handle it in some way.

Analogy: professor in a lecture; student has a question.

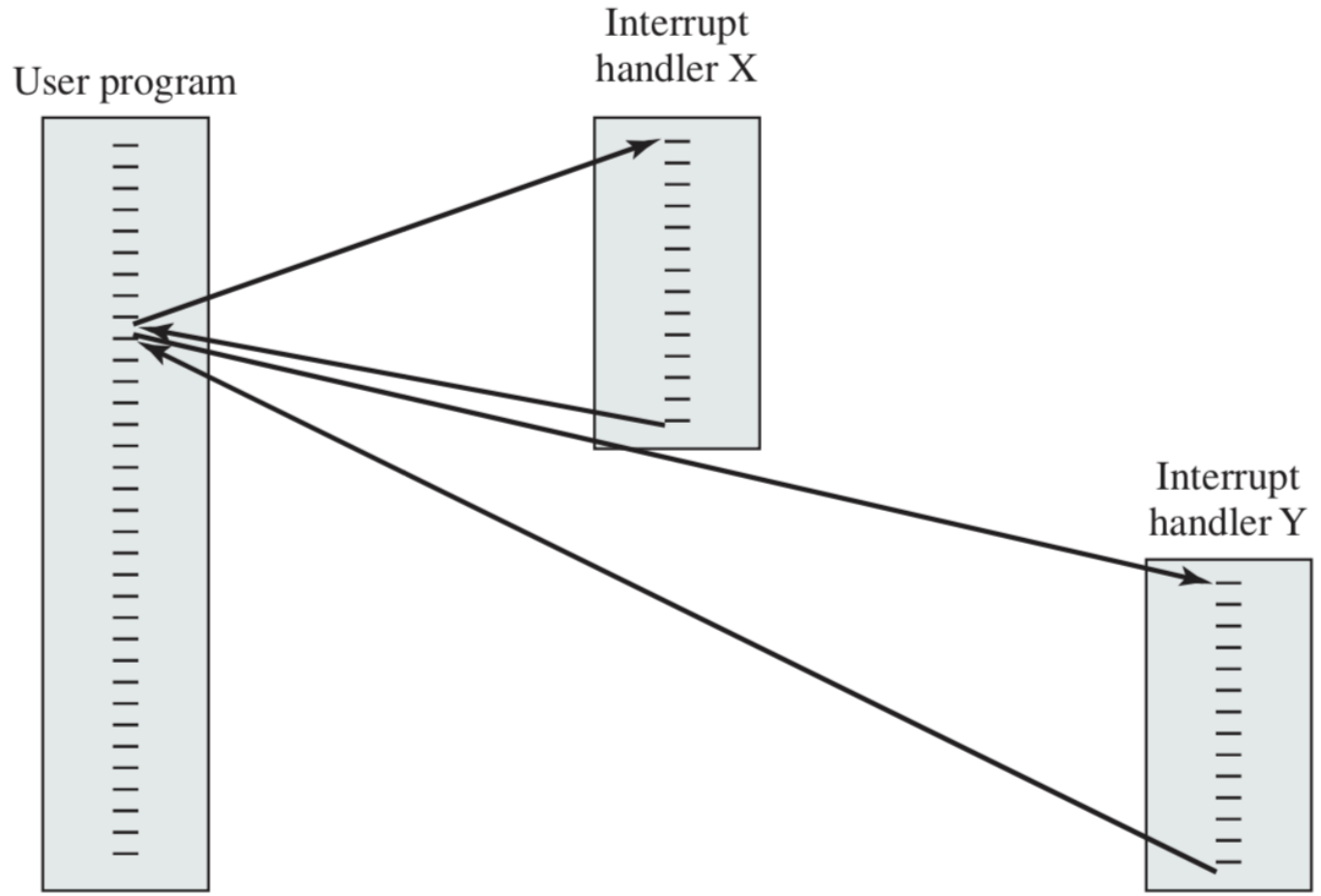
Interrupts

The OS: stores the state, handles the interrupt, and restores the state.

Sometimes the CPU is in the middle of something uninterruptible.
    Interrupts may be disabled (temporarily).

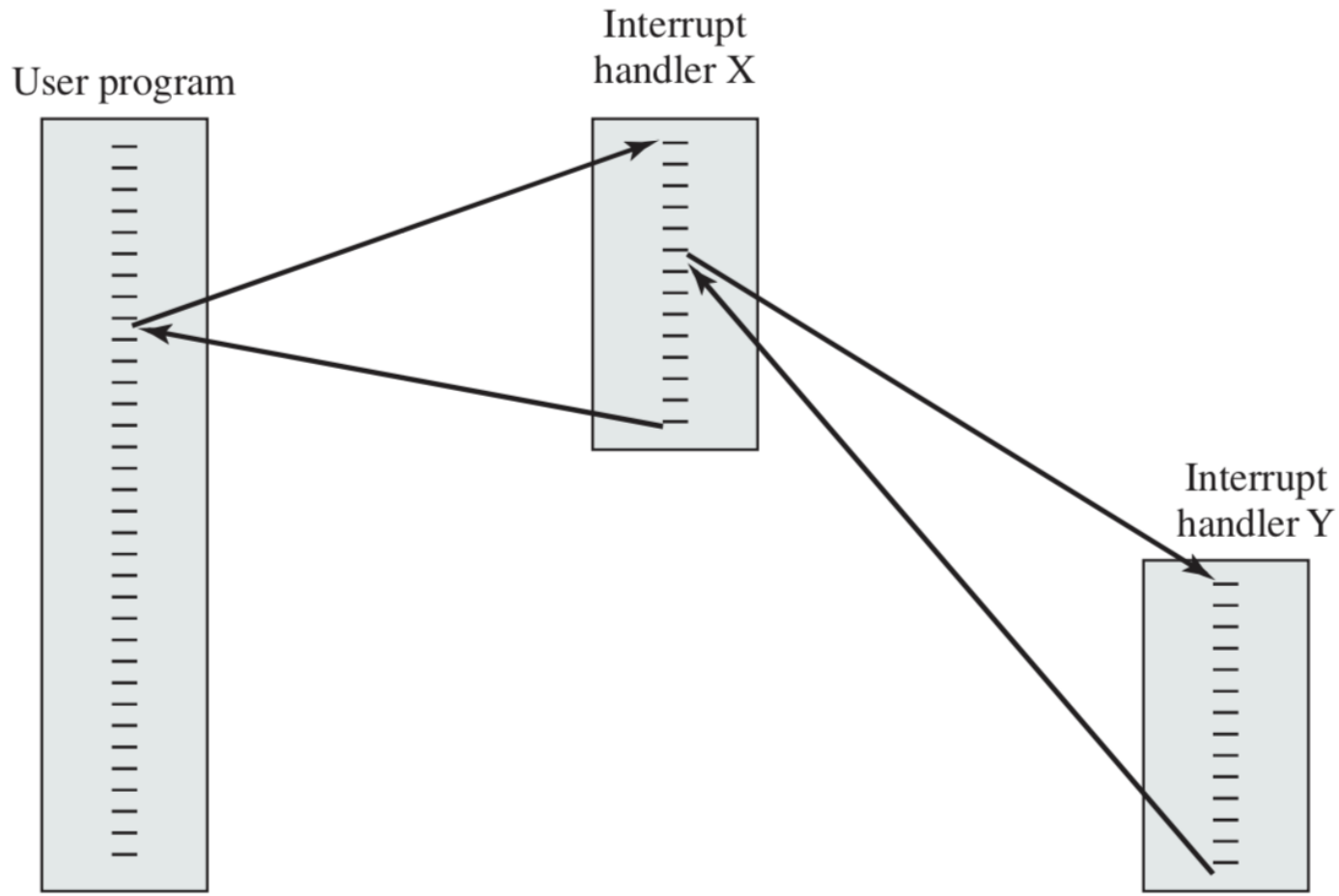
Interrupts can have different priorities.

We may also have multiple interrupts in a short period of time:



(a) Sequential interrupt processing

They may be sequential…



(b) Nested interrupt processing

Or nested. Or a combination.

Storing and Restoring State

The OS must store the program state when an interrupt occurs.

The state must be stored.

State: values of registers.

Push them onto the stack.

Interrupt finished: restore the state (pop off the stack).

Then execution continues.

That is saving and restoring the same program.

Why not restore a different program?

Scheduling decision!

Invoking System Calls

Some services run automatically, without user intervention.

In other cases, we want specifically to invoke them. How?

Admiral Ackbar Says:

Trap

Operating systems run on the basis of interrupts.

A trap is a software-generated interrupt.

Generated by an error (invalid instruction) or user program request.

Trap

If it is an error, the OS will decide what to do.

Usual strategy: give the error to the program.

The program can decide what to do if it can handle it.

Often times, the program doesn’t handle it and just dies.

User Mode and Kernel Mode

Already we saw user mode vs. supervisor (kernel) mode instructions.

Supervisor mode allows all instructions and operations.

Even something seemingly simple like reading from disk or writing to console output requires privileged instructions.

These are common operations, but they involve the OS every time.

User Mode and Kernel Mode

Modern processors track what mode they are in with the mode bit.

At boot up, the computer starts up in kernel mode as the operating system is started and loaded.

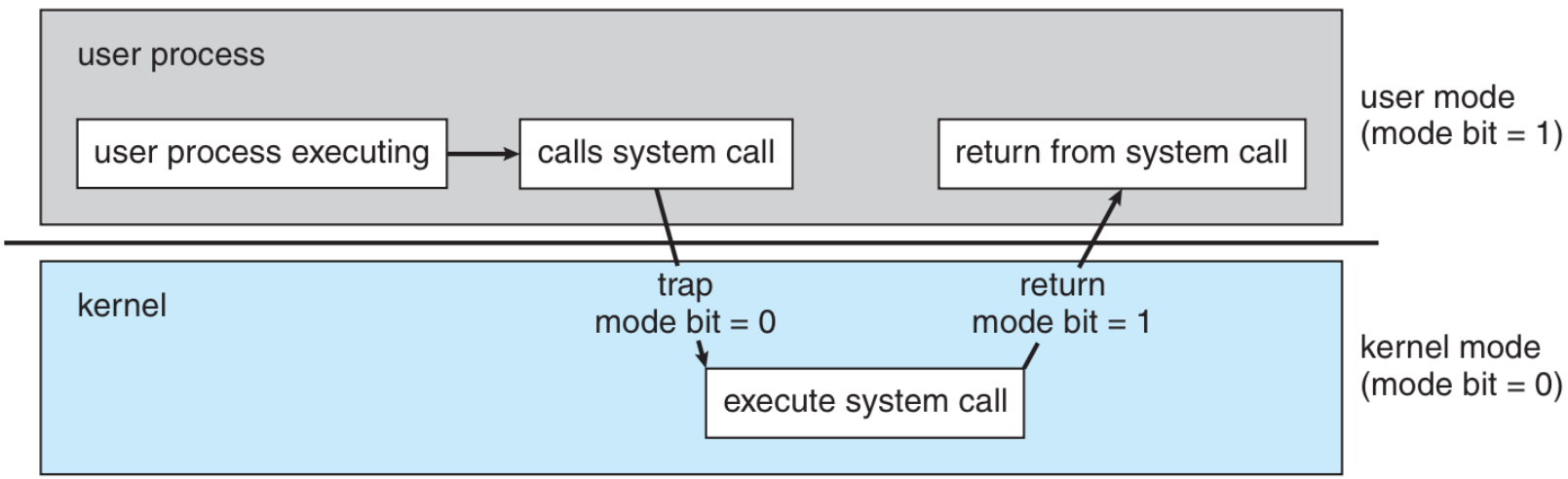
User programs are always started in user mode.

When a trap or interrupt occurs, and the operating system takes over, the mode bit is set to kernel mode.

When it is finished the system goes back to user mode before the user program resumes.

Example: Text Editor Printing

Suppose a text editor wants to output data to a printer.



User Mode and Kernel Mode: Motivation

Why do we have user and supervisor modes, anyway?

Uncle Ben to Spiderman:

User Mode and Kernel Mode: Motivation

Actually, though: “with great power comes great responsibility”.

Same as why we have user accounts and administrator accounts.

To protect the system & its integrity against errant and malicious users.

User Mode and Kernel Mode: Motivation

Multiple programs might be trying to use the same I/O device at once.

Program 1 tries to read from disk. This takes some time.

If Program 2 wants to read from the same disk, the operating system forces Program 2 to wait its turn.

Without the OS, it would be up to the author(s) of Program 2 to check and wait patiently for it to become available.

Works if everyone plays nicely.

Without enforcement of the rules, a program will do something nasty.

User Mode and Kernel Mode: Motivation

There is a definite performance trade-off.

Switching from user to kernel mode takes time.

The performance hit is worth it for the security.

UNIX Example: Reading from Disk

Here's the function we use for reading data from a file:

```
ssize_t read( int file_descriptor, void *buffer, size_t count );
```

`read` takes three parameters:

1. the file (a file descriptor, from a previous call to open);
2. where to read the data to; and
3. how many bytes to read.

Example:

```
int bytesRead = read( file, buffer, numBytes );
```

Note that `read` returns the number of bytes successfully read.

They Elected Me To Lead, Not To Read

This is a system call, and system calls have documentation.

Finding and reading this information is a key skill for systems programming.

Google (or other search engine of your choice) is your friend.

Good sources: man7.org, linux.die.net, or the website of the code library!

Example: Reading from Disk

In preparation for a call to `read` the parameters are pushed on the stack.
This is the normal way in which a procedure is called in C(++).

`read` is called; the normal instruction to enter another function.

The `read` function will put its identifier in a predefined location.

Then it executes the `trap` instruction, activating the OS.

Example: Reading from Disk

The OS takes over and control switches to kernel mode.

Control transfers to a predefined memory location within the kernel.

The trap handler examines the request: it checks the identifier.

Now it knows what system call request handler should execute: read.

That routine executes.

When it is finished, control will be returned to the `read` function.
Exit the kernel and return to user mode.

`read` finishes and returns, and control goes back to the user program.

System Call Complex Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>

void readfile( int fd );

int main( int argc, char** argv ) {
    if ( argc != 2 ) {
        printf("Usage: %s <filename>\n", argv[0]);
        return -1;
    }
    int fd = open( argv[1], O_RDONLY );
    if ( fd == -1 ) {
        printf("Unable to open file! %s is invalid name?\n", argv[1] );
        return -1;
    }
    readfile( fd );
    close( fd );
    return 0;
}
```

System Call Complex Example

```
void readfile( int fd ) {
    int buf_size = 256;
    char* buffer = malloc( buf_size );
    while ( 1 ) {
        memset( buffer, 0, buf_size );
        int bytes_read = read( fd, buffer, buf_size - 1 );
        if ( bytes_read == 0 ) {
            break;
        }
        printf("%s", buffer);
    }
    printf("\nEnd of File.\n");
    free( buffer );
}
```

System Call Summary

The steps, arranged chronologically, when invoking a system call are:

1. The user program pushes arguments onto the stack.
2. The user program invokes the system call.
3. The system call puts its identifier in the designated location.
4. The system call issues the `trap` instruction.
5. The OS responds to the interrupt and examines the identifier in the designated location.
6. The OS runs the system call handler that matches the identifier.
7. When the handler is finished, control exits the kernel and goes back to the system call (in user mode).
8. The system call returns control to the user program.

What about in the Cortex M4?

The Cortex uses the function SVC.

Other systems like x86 have different ones, and sometimes multiple.

User programs don't usually call SVC directly, but instead call an API and that wraps the system call.

Example: In Linux we call `malloc()` from `stdlib.h`, but it may call the `brk` system call (and this contains the trap).

Cortex Syscall Framework

The SVC instruction takes one integer in the range `0 - 255`.

This argument is the “identifier” in the “predefined location” from earlier.

We get to choose the meanings of the numbers:

- 0 = print something
- 1 = switch threads
- 2 = wait for input
- …

Now What

Next: Run the `SVC_Handler` (Assembly Code):

Extract the integer parameter, invoke the C function

Then: Run `SVC_Handler_Main` (C Code):

Call the individual system call corresponding to the parameter.

Activating the System Call

Inside our C code, use embedded assembly. To invoke system call 1:

```
__ASM("SVC #1");
```

The # is required!

Remember: we choose the number and what it does on the other side.

Don’t write this down, but here’s the SVC_Handler ASM:

```
AREA handle_pend,CODE,READONLY
GLOBAL SVC_Handler
PRESERVE8

SVC_Handler
    ;We will be calling this function to handle the various system calls
    EXTERN SVC_Handler_Main

    ;Check the magic value stored in LR to determine whether we are in thread mode
    TST LR,#4 ;check the magic value
    ITE EQ ; If–Then–Else
    MRSEQ r0, MSP ; If LR was called using MSP store MSP’s value in r0
    MRSNE r0, PSP ; Else store PSP’s value

    BL SVC_Handler_Main ;Jump to the C function which handles the system calls

    END
```

Start of the C Function

Warning: we're about to do some weird things to the stack.

```
void SVC_Handler_Main( uint32_t *svc_args) {
    /* ARM sets up the stack frame so that the syscall number is two
       characters behind the input argument */
    char call = ((char*)svc_args[6])[-2];

    /* Now decide what to do with call (if or switch statement)
       eg if ( call == 0 ) { ... } */

}
```

Now Wait a Minute

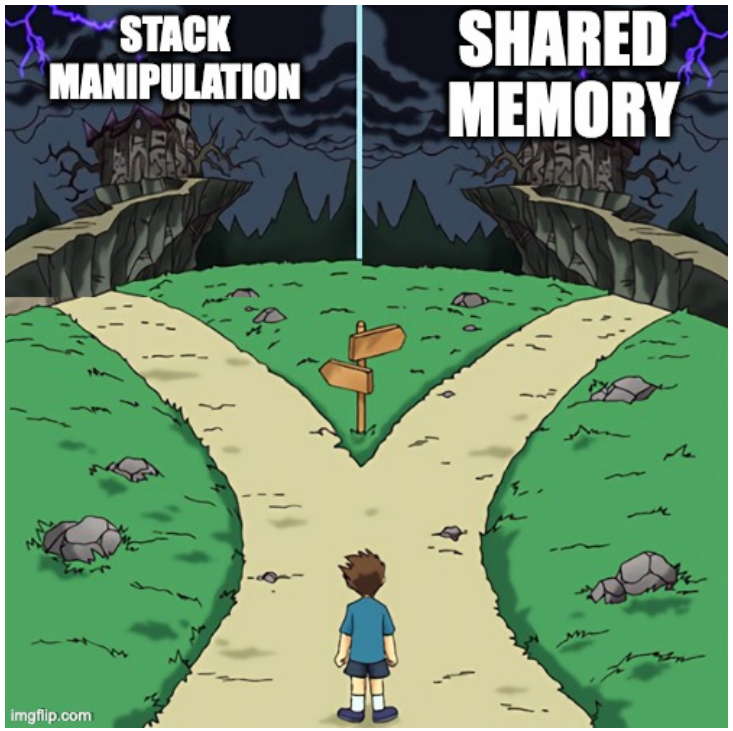
This framework is very basic, and is missing a few things.

There's no way to pass arguments from user programs to system calls…
And no way to return arguments either.

The introduction to the UNIX system call suggested how to do this (stack).

For now, we'll use shared memory areas.

Shared Memory?



Shared memory areas are a headache too.

Shared memory areas must be limited in size and protected (1 per thread).

Does it Work?

We need to get to Kernel mode and SVC is an interrupt.

Interrupts run in kernel mode!

Our rudimentary implementation is very insecure.

For fun: try to hack it: can you inject code to a syscall or hijack shared memory?