

Coroutines and Asynchronous Programming

Advanced Systems Programming (H)

Lecture 8

Lecture Outline

- Motivation
- Coroutines, **async**, and **await**
- Design patterns for asynchronous code

Motivation

- How to overlap I/O and computation?
 - Multi-threading
 - Non-blocking I/O and `select()`
- Is there a better way?

Blocking I/O

```
fn read_exact<T: Read>(input: &mut T, buf: &mut [u8]) -> Result<(), std::io::Error> {  
    let mut cursor = 0;  
    while cursor < buf.len() {  
        cursor += input.read(&mut buf[cursor..])?;  
    }  
}
```

- **Desirable to perform I/O concurrently to other operations**
 - I/O operations are slow
 - Need to wait for network, disk, etc. – operations can take millions of cycles
 - I/O operations block the thread
 - Disrupts the user experience and prevents other computations
- Want to overlap I/O and computation
- Want to allow multiple concurrent I/O operations

Blocking I/O using Multiple Threads (1/2)

- Traditionally solved by moving blocking operations into separate threads:
 - Spawn dedicated threads to perform I/O operations concurrently
 - Re-join main thread/pass back result as message once complete
- Advantages:
 - Simple
 - No new language or runtime features
 - Don't have to change the way we do I/O
 - Do have to move I/O to a separate thread, communicate and synchronise
 - Concurrent code can run in parallel if the system has multiple cores
 - Safe, if using Rust, due to ownership rules preventing data races

```
fn main() {  
    ...  
    let (tx, rx) = channel();  
    thread::spawn(move || {  
        ..perform I/O..  
        tx.send(results);  
    });  
    ...  
    let data = rx.recv();  
    ...  
}
```

Blocking I/O using Multiple Threads (2/2)

- Traditionally solved by moving blocking operations into separate threads:
 - Spawn dedicated threads to perform I/O operations concurrently
 - Re-join main thread/pass back result as message once complete
- Disadvantages:
 - Complex
 - Requires partitioning the application into multiple threads
 - Resource heavy
 - Each thread has its own stack
 - Context switch overheads
 - Parallelism offers limited benefits for I/O
 - Threads performing I/O often spend majority of time blocked
 - Wasteful to start a new thread that spends most of its time doing nothing

```
fn main() {  
    ...  
    let (tx, rx) = channel();  
    thread::spawn(move || {  
        ..perform I/O..  
        tx.send(results);  
    });  
    ...  
    let data = rx.recv();  
    ...  
}
```

Non-blocking I/O and Polling (1/4)

- Blocking I/O using threads is problematic:
 - Threads provide concurrent I/O abstraction, but with high overhead
 - Multithreading **can** be inexpensive → Erlang
 - But has high overhead on general purpose operating systems
 - Higher context switch overhead due to security requirements
 - Higher memory overhead due to separate stack
 - Higher overhead due to greater isolation, preemptive scheduling
 - Limited opportunities for parallelism with I/O bound code
 - Threads **can** be scheduled in parallel, but to little benefit unless CPU bound

Non-blocking I/O and Polling (2/4)

- Lightweight alternative: multiplex I/O operations within a single thread
- I/O operations complete asynchronously – why have threads block for them?
- Provide a mechanism to start asynchronous I/O and poll the kernel for I/O events – all within a single application thread
 - Start an I/O operation
 - Periodically poll for progress of the I/O operation
 - If new data is available, a send operation has completed, or an error has occurred, then invoke the handler for that operation

Non-blocking I/O and Polling (3/4)

- Mechanisms for polling I/O for readiness
 - Berkeley Sockets API **select()** function in C
 - Or higher-performance, but less portable, variants such as **epoll** (Linux/Android), **kqueue** (FreeBSD/macOS/iOS), I/O completion ports (Windows)
 - Libraries such as **libevent**, **libev**, or **libuv** – common API for such system services
 - Rust **mio** library
- Key functionality:
 - Trigger non-blocking I/O operations: **read()** or **write()** to files, sockets, etc.
 - Poll kernel to check for readable or writable data, or if errors are outstanding
 - Efficient and only requires a single thread, but requires code restructuring to avoid blocking

Non-blocking I/O and Polling (4/4)

- Berkeley Sockets API `select()` function in C:

```
FD_ZERO(&rfd);
FD_SET(fd1, &rfd);
FD_SET(fd2, &rfd);

tv.tv_sec = 5; // Timeout
tv.tv_usec = 0;

int rc = select(1, &rfd, &wfd, &efd, &tv);
if (rc < 0) {
    ... handle error
} else if (rc == 0) {
    ... handle timeout
} else {
    if (FD_ISSET(fd1, &rfd)) {
        ... data available to read() on fd1
    }
    if (FD_ISSET(fd2, &rfd)) {
        ... data available to read() on fd2
    }
    ...
}
```

`select()` polls a set of file descriptors for their readiness to `read()`, `write()`, or to deliver errors

`FD_ISSET()` checks particular file descriptor for readiness after `select()`

Low-level API well-suited to C programming; other libraries/languages provide comparable features

Alternatives to Non-blocking I/O?

- Non-blocking I/O can be highly efficient
 - Single thread can handle multiple I/O sources (sockets, file descriptors) at once
- But – requires significant re-write of application code
 - Non-blocking I/O
 - Polling of I/O sources
 - Re-assembly of data
- Can we get the efficiency of non-blocking I/O in a more usable manner?
 - Maybe – coroutines and asynchronous code

Motivation

- How to overlap I/O and computation?
 - Multi-threading
 - Non-blocking I/O and `select()`
- Is there a better way?