Coroutines and Asynchronous Programming

Advanced Systems Programming (M)
Lecture 7
Lecture Outline

• Motivation
• `async` and `await`
• Design patterns for asynchronous code
• An asynchronous future?
Motivation

- Blocking I/O
- Multi-threading → overheads
- `select()` → complex
- Coroutines and asynchronous code
Blocking I/O

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

```rust
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..])?;
    }
}
```
Concurrent I/O using Multiple Threads (1/2)

- Move 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

```rust
def main() {
    ... let (tx, rx) = channel();
    thread::spawn(move|| {
        ...perform I/O...
        tx.send(results);
    });
    ...
    let data = rx.recv();
    ...
}
```
Concurrent I/O using Multiple Threads (2/2)

- Move 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

```rust
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)

• 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 writeable 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

```c
FD_ZERO(&rfds);
FD_SET(fd1, &rfds);
FD_SET(fd2, &rfds);

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

int rc = select(1, &rfds, &wfds, &efds, &tv);
if (rc < 0) {
    ... handle error
} else if (rc == 0) {
    ... handle timeout
} else {
    if (FD_ISSET(fd1, &rfds)) {
        ... data available to read() on fd1
    }
    if (FD_ISSET(fd2, &rfds)) {
        ... 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
Coroutines and Asynchronous Code

• Non-blocking I/O can be highly efficient
  • Single thread handles multiple I/O sources at once
    • Network sockets
    • File descriptors
  • Or application can partition I/O sources across a thread pool
• 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?
Coroutines and Asynchronous Code

- Provide language and run-time support for I/O multiplexing on a single thread, in a more natural style

```rust
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..])?
    }
}
```

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

- Runtime schedules `async` functions on a thread pool, yielding to other code on `await!()` calls → low-overhead concurrent I/O
async and await

- Coroutines and asynchronous code
- Runtime support requirements
- Benefits and trade-offs
Programming Model

- Structure I/O-based code as a set of concurrent coroutines that accept data from I/O sources and yield in place of blocking

What is a coroutine?

A generator yields a sequence of values:

```python
def countdown(n):
    while n > 0:
        yield n
        n -= 1
```

```python
>>> for i in countdown(5):
...   print i,
...
5 4 3 2 1
>>> 
```

A function that can repeatedly run, yielding a sequence of values, while maintaining internal state

Calling `countdown(5)` produces a generator object. The `for` loop protocol calls `next()` on that object, causing it to execute until the next `yield` statement and return the yielded value.

→ Heap allocated; maintains state; executes only in response to external stimulus

Based on: http://www.dabeaz.com/coroutines/Coroutines.pdf
Programming Model

- Structure I/O-based code as a set of concurrent coroutines that accept data from I/O sources and yield in place of blocking

What is a coroutine?

A coroutine more generally consumes and yields values:

```python
def grep(pattern):
    print "Looking for %s" % pattern
    while True:
        line = (yield)
        if pattern in line:
            print line

>>> g = grep("python")
>>> g.next()
Looking for python
>>> g.send("Yeah, but no, but yeah, but no")
>>> g.send("A series of tubes")
>>> g.send("python generators rock!")
python generators rock!
```
What is a coroutine?

A coroutine is a function that executes *concurrently* to – but not in parallel with – the rest of the code.

It is event driven, and can accept and return values.
Programming Model

• Structure I/O-based code as a set of concurrent coroutines that accept data from I/O sources and yield in place of blocking

  • An async function is a coroutine
    • Blocking I/O operations are labelled in the code – await – and cause control to pass to another coroutine while the I/O is performed

• Provides concurrency without parallelism
  • Coroutines operate concurrently, but typically within a single thread
  • await passes control to another coroutine, and schedules a later wake-up for when the awaited operation completes
  • Encodes down to a state machine with calls to select(), or similar

• Mimics structure of code with multi-threaded I/O – within a single thread
async Functions

- An **async** function is one that can act as a coroutine
  - It is executed *asynchronously* by the runtime
  - Widely supported – Python 3, JavaScript, C#, Rust *(in progress)*, …

```python
#!/usr/bin/env python3
import asyncio

async def fetch_html(url: str, session: ClientSession) -> str:
    resp = await session.request(method="GET", url=url)
    html = await resp.text()
    return html

asyncio.run(async function)
```

- Main program must trigger asynchronous execution by the runtime:
  - Starts asynchronous polling runtime, runs until specified **async** function completes
  - Runtime drives **async** functions to completion and handles switching between coroutines
**await** Future Results

- An **await** operation yields from the coroutine
  - Triggers an I/O operation – and adds corresponding file descriptor to set polled by the runtime
  - Puts the coroutine in queue to be woken by the runtime, when file descriptor becomes ready

```python
#!/usr/bin/env python3
import asyncio

async def fetch_html(url: str, session: ClientSession) -> str:
    resp = await session.request(method="GET", url=url)
    html = await resp.text()
    return html
...```

- If another coroutine is ready to execute then schedule wake-up once the I/O completes, and pass control passes to the other coroutine; else runtime blocks until either this, or some other, I/O operation becomes ready
- At some later time the file descriptor becomes ready and the runtime reschedules the coroutine – the I/O completes and the execution continues
**async and await programming model**

- Resulting asynchronous code should follow structure of synchronous (blocking) code:

```rust
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..])?
    }
}
```

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

- Asynchronous code needs runtime support to execute the coroutines and poll the I/O sources for activity

- An `async` function that returns data of type `T` compiles to a regular function that returns `impl Future<Output=T>`

```
pub enum Poll<T> { Ready(T), Pending, }
pub trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, lw: &LocalWaker) -> Poll<Self::Output>;
}
```

- i.e., it returns a `Future` value that represents a value that will become available later
- The runtime continually calls `poll()` on `Future` values until all are `Ready`
  - A future returns `Ready` when complete
  - A future returns `Pending` when blocked on `await`ing some I/O operation
  - Calling `tokio::run(future)` starts the runtime

- Well supported in Python 3 and JavaScript – the runtime for Rust is still experimental (https://tokio.rs/)
Design Patterns for Asynchronous Code

- Compose *Future* values
- Avoid blocking I/O
- Avoid long-running computations
Compose **Future** Values

- **async** functions should be small, limited scope
- Perform a single well-defined task:
  - Read and parse a file
  - Read, process, and respond to a network request

- Rust provides combinators that can combine **Future** values, to produce a new **Future**:
  - `for_each()`, `and_then()`, `read_exact()`, `select()`
  - Can ease composition of asynchronous functions – but can also obfuscate
Avoid Blocking Operations

- Asynchronous code multiplexes I/O operations on single thread
  - Provides asynchronous aware versions of I/O operations
    - File I/O, network I/O (TCP, UDP, Unix sockets)
    - Non-blocking, return `Future` values that interact with the runtime
  - Does **not** interact well with blocking I/O
    - A `Future` that blocks on I/O will block **entire** runtime

- Programmer discipline required to ensure asynchronous and blocking I/O are not mixed within a code base
  - Including within library functions, etc.
Avoid Long-running Computations

• Control passing between Future values is explicit
  • await calls switch control back to the runtime
  • Next runnable Future is then scheduled
  • A Future that doesn’t call await, and instead performs some long-running computation, will starve other tasks

• Programmer discipline required to spawn separate threads for long-running computations
  • Communicate with these via message passing – that can be scheduled within a Future
An Asynchronous Future?

- Is asynchronous code a good idea?
When to use Asynchronous I/O?

• **async/await** restructure code to efficiently multiplex large numbers of I/O operations on a single thread
  • Gives a *very natural programming model when each task is I/O bound*
    • Code to perform asynchronous, non-blocking, I/O is structured very similarly to code that uses blocking I/O operations
    • Runtime can schedule many tasks can run concurrently on a single thread
    • Each task is largely blocked awaiting I/O – little overheads
When to use Asynchronous I/O?

• **async/await** restructure code to efficiently multiplex large numbers of I/O operations on a single thread

• **Problematic** with blocking operations
  - If a task performs a blocking operation, it locks the entire runtime – all potentially blocking calls must be updated to use asynchronous I/O operations
  - Potential to fragment the library ecosystem

• **Problematic** with long-running computations
  - Long-running computations starve other tasks of CPU time – runtime only switches between tasks when an asynchronous operation is started
  - Need to insert context switch calls – isn’t this just **cooperative multitasking** reimagined?
    - Windows 3.1, MacOS System 7
Performance of Asynchronous I/O

• Do you **really** need asynchronous I/O?
  
  • Threads are more expensive than `async` functions and tasks in a runtime
  
  • But threads are not **that** expensive – a properly configured modern machine can run thousands of threads
    
    • ~2,200 threads running on the Core i5 laptop these slides were prepared on, in normal use
    
    • Varnish web cache ([https://varnish-cache.org](https://varnish-cache.org)): “it’s common to operate with 500 to 1000 threads minimum” but they “rarely recommend running with more than 5000 threads”
    
    • Unless you’re doing something **very** unusual you can likely just spawn a thread, or use a pre-configured thread pool, and perform blocking I/O – and communicate using channels, **even if this means spawning thousands of threads**

• Asynchronous I/O **can** give a performance benefit
  
  • But this performance benefit will usually be small
  
  • Choose asynchronous programming because you prefer the programming style, accepting that it will often not significantly improve performance
Summary

- Blocking I/O
  - Multi-threading → overheads
  - `select()` → complex
- Coroutines and asynchronous code
- Is it worth it?