University of Glasgow | School of Computing Science

Assessed Coursework

Course Name	**Advanced Systems Programming (H/M)**			
Coursework Number	**Exercise 2**			
Deadline	**Time:**	**4:30pm**	**Date:**	**10 March 2025**
% Contribution to final course mark	**10%**			
Solo or Group ✓	**Solo**	✓	**Group**	
Anticipated Hours	**10**			
Submission Instructions	**Submit via Moodle**			
Please Note: This Coursework cannot be Re-Assessed				

Code of Assessment Rules for Coursework Submission

Deadlines for the submission of coursework which is to be formally assessed will be published in course documentation, and work which is submitted later than the deadline will be subject to penalty as set out below.

The primary grade and secondary band awarded for coursework which is submitted after the published deadline will be calculated as follows:

(i) in respect of work submitted not more than five working days after the deadline
   a. the work will be assessed in the usual way;
   b. the primary grade and secondary band so determined will then be reduced by two secondary bands for each working day (or part of a working day) the work was submitted late.

(ii) work submitted more than five working days after the deadline will be awarded Grade H.

Penalties for late submission of coursework will not be imposed if good cause is established for the late submission. You should submit documents supporting good cause via MyCampus.

Penalty for non-adherence to Submission Instructions is 2 bands

You must complete an "Own Work" form via https://studentltc.dcs.gla.ac.uk/ for all coursework

Advanced Systems Programming (H/M) 2024-2025 – Exercise 2

School of Computing Science, University of Glasgow

24 February 2025

Introduction

The Advanced Systems Programming (H/M) course uses the Rust programming language (`https://rust-lang.org/`) to illustrate several advanced topics in systems programming. One such topic is concurrent programming, since a key feature needed to make effective use of a modern multicore processor is safe concurrency.

This exercise explores concurrent programming in Rust. **This is an assessed exercise that is worth 10% of the marks for this course.**

Networked Applications and the DNS

Networked applications frequently operate in a client-server manner. Clients connect to the server, make a request, wait for a response to be received, and then disconnect. The server is generally identified by a domain name, such as `www.glasgow.ac.uk`, and clients must perform a DNS lookup to resolve that domain name to an IP address before they can establish a connection to the server.

A DNS lookup can return multiple IP addresses for a domain name. This can occur when multiple hosts serve requests for a popular service, spreading the load between them, or when a server is reachable using both IPv4 and IPv6.

Write a program, `dnslookup`, using the Rust programming language, that takes a list of one or more domain names on the command line, and performs a DNS lookup for each name. After each DNS lookup, your program should print out the list of IP addresses returned, with each address prefixed with the domain name and address type. For example, if asked to resolve the name `www.google.com`, your program might print the following:

```
$ ./dnslookup www.google.com
www.google.com IPv6 2a00:1450:4009:801::2004
www.google.com IPv4 172.217.23.36
$
```

The student Linux servers (`stlinux01.dcs.gla.ac.uk` to `stlinux08.dcs.gla.ac.uk`) have IPv6 enabled, and are able to resolve IPv6 addresses. Rust provides implementations of the trait `std::net::ToSocketAddrs` in the standard library that can be used to perform DNS lookups; you do not need to use an external crate.

This part of the exercise is preparatory work. You do not need to submit the code for your `dnslookup` program, and it is not assessed.

Making Sequential Connections

Once a client has resolved the domain name for the server into a list of IP addresses, it tries to establish a connection. The way this is typically taught is that the client assumes the DNS lookup returns the addresses in order of preference, and tries to connect to each address in turn, stopping when a connection is successfully established.

The `dnslookup` example given previously showed an IPv6 and an IPv4 address being returned from the DNS lookup. In this case, the client would first try to connect to the IPv6 address of the server, since this was first in the list of addresses returned by the DNS lookup. Then, if the connection to the IPv6 address failed, it would try to connect to the IPv4 address.

Using the Rust programming language, write a program, `seqcon`, to perform a DNS lookup for a domain name given on the command line, and to establish a connection that server. Your program should loop over the list of IP addresses returned from the DNS lookup, and try to connect to each address in turn on TCP port 80 (the HTTP port). This should continue until either a connection is successfully established, or all of the IP addresses of the server have been tried.

Once it successfully establishes a connection, your program should print of the IP address to which it connected. Then, it should send the following text to the server:

```
GET / HTTP/1.1
Host: hostname
```

replacing `hostname` with the DNS name of the host to which it has made the connection. Each line *must* end with a carriage return and newline ("`\r\n`"). There *must* be a blank line at the end of the request. Following this, the server should read and print all data returned from the server until the connection is closed by the server. It should then close the connection and exit. The `seqcon` program is a trivial HTTP client: it fetches and displays the main page of the specified web server.

It is likely that the code you wrote for the `dnslookup` program will be useful when developing the `seqcon` program.

This part of the exercise is preparatory work. You do not need to submit your `seqcon` program, and it is not assessed.

Making Concurrent Connections

The problem with trying to connect to each address of a server in turn is that it can be slow if the server is not reachable on some addresses. For example, in some cases it can take tens of seconds for a connection request to timeout if blocked by a firewall. A better approach is to try to connect to all the addresses concurrently, proceed with the first that successfully connects, and close all the other connections. This uses

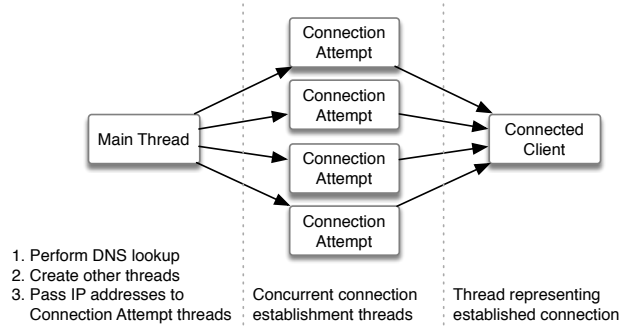


Figure 1: Concurrent connection architecture

more resources, as it attempts to open multiple connections in parallel, but can be much faster to connect.

Using the Rust programming language, write a program, `concon`, to perform a DNS lookup for a domain name given on the command line, and to establish a connection to that server using concurrent connection requests. The `concon` program should be structured using three types of thread that communicate using `std::sync::mpsc::channel` channels, as shown in Figure 1:

- The **main thread** should perform the DNS lookup for the domain name specified on the command line. It should create a single connected client thread, plus a connection attempt thread for each IP address returned by the DNS lookup. These threads should be connected using MPSC channels as shown in Figure 1. *After all the threads have been created*, the main thread should send a message to each connection attempt thread instructing it to attempt to connect to one of the IP addresses returned from the DNS lookup.
- Each of the **connection attempt threads** listens on the channel connecting it to the main thread, waiting to receive a message containing an IP address to which it should connect. When such a message is received, it tries to connect to port 80 on that address using `std::net::TcpStream::connect`. If the connection succeeds, it sends the resulting `TcpStream` object down the channel to the connected client thread and then exits. If the connection does not succeed, it simply exits.
- The **connected client thread** waits for connections (i.e., `TcpStream` objects) sent from the connection attempt threads. The connected client thread prints the peer address of the first `TcpStream` it receives, then sends a GET request to the server using that TCP stream, prints the data returned, and closes the connection (as was done in the `seqcon` program). For all other connections, the connected client thread simply closes the connection without sending a GET request.

The `concon` program will use a separate channel to connect the main thread to each connection attempt thread, and a single channel (where the transmit side has been duplicated using `clone()`) to link the connection attempt threads to the connected client thread.

The goal is to demonstrate the concurrency primitives in Rust, showing how threads can be spawned and how message passing can be used to pass data between threads. **You must submit your `concon` program for assessment.**

Submission

You must submit the source code for your `concon` program for assessment. Create a directory named `asp-ex2-`*`GUID`*, replacing *`GUID`* with your GUID (your student number followed by the first letter of your surname). Put a copy of the source code for your `concon` program into this directory. Do not include compiled binaries (i.e., run `cargo clean` before copying the files into the `asp-ex2-`*`GUID`* directory). Create a zip archive of the directory, as a file called `asp-ex2-`*`GUID`*`.zip`. Submit the zip archive via Moodle.

As an example, if your GUID was 1234567a, you would perform the following steps to create the zip archive, after copying your source code into the `asp-ex2-1234567a` directory:

```
zip -r asp-ex2-1234567a.zip asp-ex2-1234567a/
```

Check carefully that your zip archive extracts into the correct subdirectory, contains only the requested files, and has the correct filename.

Assessment and Marking Scheme

This exercise is worth 10% of the mark for this course. You must submit your solution before 4:30pm on 10 March 2025. Following the University code of assessment, late submissions will be accepted for up to 5 working days beyond this due date. Late submissions will receive a two band penalty for each working day, or part thereof, the submission is late. Submissions that are received more than five working days after the due date will be awarded a band of H.

Submissions that are not made via Moodle, that have the wrong filename, that have a zip archive that extracts into the wrong directory or that otherwise do not follow the submission instructions will be subject to a two band penalty. This penalty is in addition to any late submission penalty. *This penalty will be strictly enforced.* If you are ill, or have other circumstances that may affect your submission, then you may contact the student support office *before* the deadline to request an extension, following the usual procedure.

Marks will be awarded based on inspection of the source code, and on the results of compiling and running the `concon` program, as follows:

- Up to [2 marks] for compiling with no errors and no warnings using the version of the Rust compiler installed on the student Linux servers (`stlinux01` - `stlinux08`).
- Up to [8 marks] for successful operation of the program, including the ability to perform DNS lookups and to race IPv4 and IPv6 connection attempts. Your program will be run using the following commands on one of the student Linux servers:

```
unzip asp-ex2-$GUID.zip
cd asp-ex2-$GUID
cargo run $DNSNAME
```

where the variables `$GUID` and `$DNSNAME` will be set to your GUID and the domain name to look-up. Several domains will be tested, with a mixture of IPv4 and IPv6 addresses.

- Up to [10 marks] for the design and implementation of the main thread. This will include correct initialisation of the other threads, correct use of channels to communicate between the threads, correctly performing the DNS lookup, and correctly passing the IP addresses to the connection attempt threads.
- Up to [6 marks] for the design and implementation of the connection attempt threads. This will include correct handling the input and output channels, correctly establishing a connection and passing the `TcpStream` to the connected client thread.
- Up to [4 marks] for the design and implementation of the connected client thread. This will include looping to receive multiple connections from the connection attempt threads, printing the peer address of the first successful connection, sending the HTTP GET request and receiving the response, and closing connections when no longer needed.

The result will be a numeric mark out of 30. This numeric mark will be converted to a percentage, then the percentage mark will be converted to a band on the 22-point University of Glasgow scale using the standard translation table for the School of Computing Science. Any applicable penalty for late submission and/or for not following submission instructions will then be applied, and a band will be returned. A brief written justification for the band will also be supplied.

Feedback will comprise a summary of factors considered when marking, a breakdown of the marks awarded, and brief written comments.