CSCI 5103: Operating Systems
Fall 2015
Instructor: Jon Weissman
Project 1: OS Support For Concurrency
Due: Oct 26th (midnight)

1. Overview

In this project, you will implement different concurrency abstractions, threads and asynchronous I/O (event-driven programming), and compare them. This will provide insight to how well the OS supports different models of concurrency. A server, for example, can create a thread whenever it receives a request from a user such that each thread will be able to handle each request concurrently. A server may instead may use asynchronous I/O with a single thread to handle multiple requests. To see the difference between these approaches in terms of performance and implementation complexity, you will implement these different abstractions on a simple data server which receives network connections and simply retrieves the data from the requesting client. This way we can easily measure the low-level cost of the concurrency mechanisms. Before starting the project, we encourage you to read pages 170-176 in the book as background.

2. Project Details

You will implement a simple server which receives network connections and reads request data concurrently from a set of clients. A client will connect to the server through a server (main) socket. Once the server gets a connection, it creates a new socket (through `accept`) and the client will write/send data through the new socket. The data sent by a clients can be any data but of a specific (or variable) size that you control. To reduce performance interference, the server will not store data or do much of anything but only retrieve the input data from the clients. Once the server receives data from a socket completely, it cleans up any resources associated with the socket (connection and data). Your server will implement different the following concurrency abstractions.

1) **Threads**

   When a new connection is accepted, the server will create a new thread and the thread will retrieve client data. Remember to `join` or `detach` threads.

2) **Asynchronous I/O + *Polling***

   The server will handle all I/O operations (connections and receiving data) with a single thread using asynchronous I/O and *polling*. When a new connection is accepted, the server will store the socket descriptor (in a data-structure of your choice) and then issue an asynchronous read from the socket:
   a) Using `aio_read` which will return immediately. To see whether the read operation is done or not for each socket descriptor, the server will keep checking them one by one by *polling*.
   b) Using `read` but with control flags set to make it asynchronous (see `fnctl`) which will return immediately. This is the older Unix style.

3) **Asynchronous I/O + *Select()***

Similar to 2) implementation except use `select` instead of *polling*. In this implementation, you will use `select` which will be blocked until any I/O (connection or data) is ready. Once `select` returns, your server will be able to accept the connection or receive data or both. In this implementation you will use asynchronous `read` instead of `aio_read`.

For 2) Asynchronous I/O, you will need to make the listening socket non-blocking to avoid being blocked while waiting a connection with `accept`. For Asynchronous I/O, the server will receive data by calling `aio_read` or `read` several times to receive the full amount of the data sent as needed.

- For using `aio_read`, you will need to pass buffer address where the data will be written in the background but the server may have no idea about the size of incoming data. To read all data completely, you will need call `aio_read` several times with appropriate parameters (offset and length). Even in the case where you know the incoming data size beforehand, passing in the exact size of the buffer may not be a good idea if the incoming data size is huge (this is even worse when data is coming from multiple clients concurrently).
- For 3) implementation, this is required to prevent an incoming data from monopolizing overall running time. For example, if there are multiple incoming data streams and some of them are large data (> 10MB) and some are small (< 100KB), the server may read at most 1MB at a time for each iteration to receive data concurrently to improve throughput.

You need to measure the time for the server to handle the incoming data requests (i.e. throughput). I'll be impressed if you can measure latency of a single request *at the server* but this is harder to measure. You can use any timer function but make sure it is fine-grained enough. You can assume that there will only one incoming data stream from each connection and there will be at most 10,000 concurrent connections with the maximum data sent less than 10MB per each connection.

You must use C/C++ and the code must run in a UNIX (Linux) environment. You are expected to have a-priori knowledge in C-Unix system programming and/or the ability the figure out how code works by investigation. We will provide a test client program which sends multiple data requests simultaneously to the server (written in python). But you can modify it or build your own program to test your server as you want.

3. Performance Comparisons

Compare implementations 1-2-3 and then 2a-2b in terms of throughput as a function of the following varying conditions:

1) Number of Requests - You will compare the performance based on the number of requests.
   - 1 ~ 10,000 at a time.
2) Various incoming data size – Clients will send various sizes of data to the server.
   - Mix of sizes. E.g. clients will send random size of data, small (1KB) – large (10MB)
   - Single varied size. E.g) Clients will send specific size of data 1KB or 100KB or 10MB
3) Different OS: Linux and Unix
   - Linux (any CS, CSE Linux machine) VS. Solaris (Unix) (caesar.cs)
4) Question: For each of the implementation methods describe the flow of control between the

application and the OS. For this you may want to read up on the system calls. For example, is the thread library user and/or kernel space?

4. Project Group

All students should work in groups of size 2 or 3. If you have difficulty in finding a partner, we encourage you to use the forum to find your partner. **Only one submission is needed for each group.**

5. Test Cases

You should also develop your own test cases for all the implementations, and provide documentation that explains how to run each of your test cases, including the expected results and an explanation of why you think the results look as they do. Please be aware of the CSE firewall if which network access to CSE (CS) machines may be blocked on certain network ports.

6. Deliverables

a. Document that contains your performance results and analysis to also include:
   Testing description, including a list of cases attempted (also must include negative cases).
   Not to exceed 3 pages.
b. Source code, makefiles and/or a script to start the system, and executables (**No object files**).

7. Grading

The grade for this assignment will include the following components:

a. 30% - The documents.
   - 30% - The report: comparison and scenarios and analysis.
b. 60% - The functionality and correctness of each implementation.
   - 20% - Threads
   - 20% - Asynchronous + *Polling*
   - 20% - Asynchronous + *Select* ()
c. 10% - The quality of the source code, in terms of style and in line documentation

The programs should be robust. You will have points deducted for any crash or exception.

8. Reference

1. AIO tutorial: “man aio” in Linux (Unix machine) or http://fwheel.net/aio.html
2. AIO http://www.informit.com/articles/article.aspx?p=607373&seqNum=4
3. Non-Blocking VS. asyncIO
   http://stackoverflow.com/questions/2625493/asynchronous-vs-non-blocking
4. Timer – gettimeofday Example
   http://www.cs.loyola.edu/~jglenn/702/S2008/Projects/P3/time.html