Agenda for Today

1. Lab Q&A
2. Concurrency with I/O
3. Live Coding
Sudoku Lab Q&A

How much memory does a board use?
Each cell is represented by a uint16_t, which is two bytes. There are 81 cells in a board. So a board is something like 162 bytes, but there may be some padding.

I’m getting an error about permissions when I run nvcc. What should I do?
There’s something wrong with torres.cs.grinnell.edu, and you’re probably on that machine. Start a new session and hope for a new machine.

We called cell_to_digit on a bunch of cells in a board, all of them seem to return zeroes. What gives?
You may not have copied the board to the GPU. Make sure you’re doing that. Most cells will actually return zero, so that’s expected.

When we call cell_to_digit, we get -1 sometimes. Why would that happen?
I think that would happen if the cell has no bits turned on (meaning no values are possible there) cell_to_digit might return -1.
Warning
A cell is a member of its own row, column, and region. But you should not propagate constraints from a cell to itself. That would lead you to say a cell has no possible values.
Another related issue: you can only propagate values from cells that have a single possible value. If cell could be 2 or 4, you can’t say anything definite about whether or not another cell in its row is 2, 4, or neither.
Concurrency with I/O
Server: One Client at a Time

A simple server accepts a connection from a client. It repeatedly reads messages from the client and prints them. When the client disconnects it repeats the whole process by waiting for a new client.

What happens if two clients try to connect at the same time?

The server will accept one client, but the second will just be stuck waiting for a connection. We don’t know which client will connect first.

Clients trying to connect to a server will queue up on the server socket waiting to be accepted.
Server: Multiple Clients at Once

How could we change the server to communicate with multiple clients at once?

The server could create a thread to talk to each client instead of talking to the clients in the main thread. The main thread would only be responsible for establishing connections. We can achieve almost the same thing with multiple processes.

What if the main thread could talk to any client, and we’d just identify them so it can talk to a specific client when we receive a message. The problem with this approach is that we have to call read() on the socket connected to a client, and that operation will block the server until that specific client sends something. We need some way to wait for messages from a bunch of sockets at the same time instead of waiting for a specific socket.
How could we change the server to communicate with multiple clients at once? We could use some sort of non-blocking read on sockets. If we had ten clients connected, we’d just loop over all ten sockets and try to read from each. Most reads would return EWOULDBLOCK instead of reading data, but that’s okay.

It could take quite a while to get to the 400th socket in this process. Every read attempt is a system call, and those can take a while. The other downside is that this program will completely overload the OS kernel with these constant system calls. This process is called polling, where we constantly check for updates across a list of channels.

There is a system call available to us called poll() that does the same thing without the performance issues. We do still have to loop over file descriptors, but each iteration of the loop isn’t issuing a system call, so it’s much better. There’s a linux-only variant called ppoll() that saves some work here. The reading covered select(), which is just an old, ugly version of poll().
Live Coding