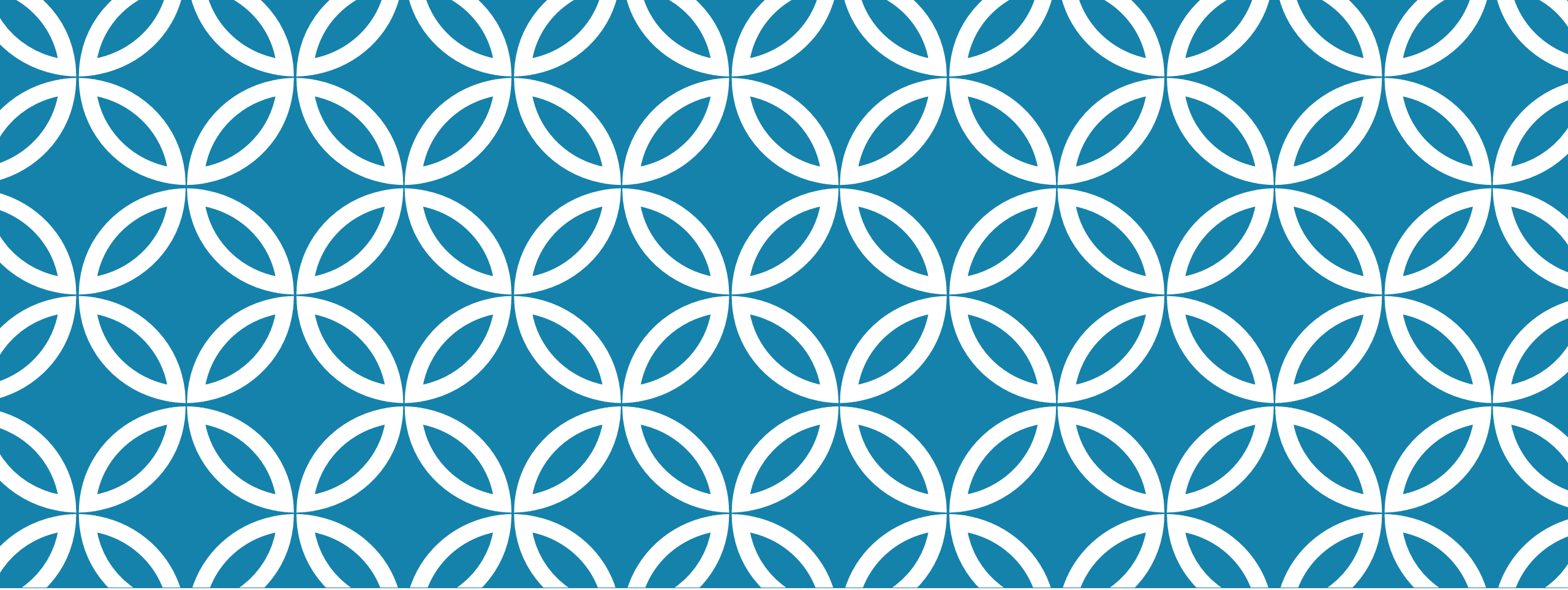


NON-BLOCKING IO TECHNIQUES

Idle hands are the Devil's playground.

STRATEGIES

Blocking Sockets (One Thread)

Non-Blocking Sockets

Blocking Sockets (Multiple Threads)

Event-driven IO

select()

Advanced Techniques

BLOCKING SOCKETS (ONE THREAD)

"Just live with it!"

Pros:

- Simple

Cons:

- Non-interactive

BLOCKING SOCKETS (MULTIPLE THREADS)

"One thread blocks, the other runs free"

Pros:

- Efficient
- The socket code is easy to write

Cons:

- Threading is hard

SELECT()

"Which of these sockets is awaiting action?"

Pros:

- Handles large numbers of sockets well

Cons:

- Complex API
- Not very efficient for one or two sockets

NON-BLOCKING SOCKETS

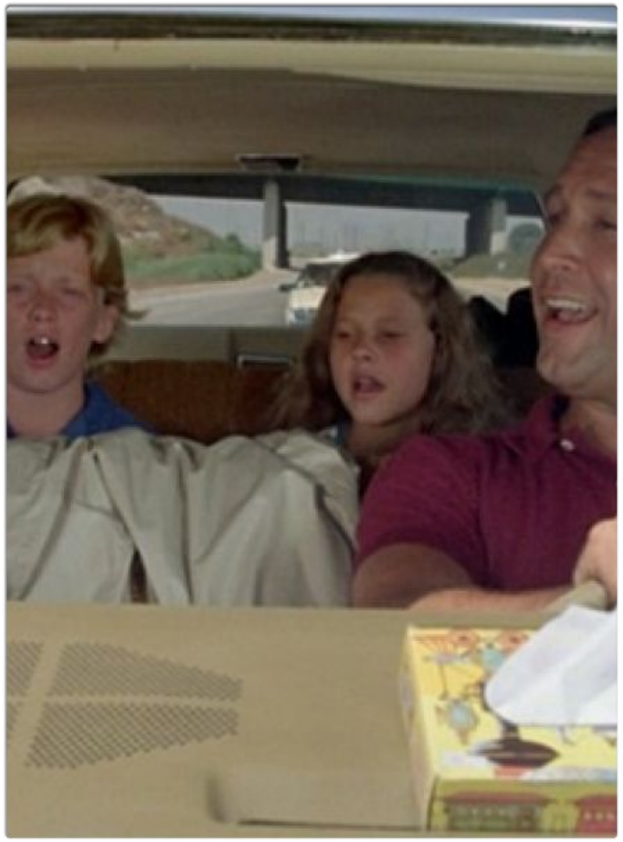
"Are we there yet? Are we there yet? Are we there yet?"

Pros:

- Simple

Cons:

- Polling isn't very efficient…
- …unless your program is already structured that way



EVENT-DRIVEN IO

"Handle mouse, handle keyboard, handle network…"

Pros:

- Simple
- Efficient

Cons:

- Platform-specific

ADVANCED TECHNIQUES

"Completion Ports and Overlapped IO"

Pros:

- Maximally efficient for large numbers of sockets

Cons:

- Maximally complex

STRATEGIES

Blocking Sockets (One Thread)

Non-Blocking Sockets

Blocking Sockets (Multiple Threads)

select()

Event driven IO

Advanced Techniques

NON-BLOCKING SOCKETS

1. Create the socket as usual
2. Tell the OS to set the “non-blocking IO” flag
3. Watch for WSAEWOULDBLOCK errors

```
int mode = 1;

int err = ioctlsocket(sock,
                      FIONBIO, &mode);

if (err != NO_ERROR)
    printf(“failure!”);
```

NON-BLOCKING SOCKETS: RECV() AND SEND()

1. Call as normal
2. Positive result = success
3. Negative result = error
   1. If WSAEWOULDBLOCK, call again!
4. For TCP recv() *only*, 0 result = socket closed

```
int bytes = recv(sock, buffer,
                 maxBytes, 0);
if (bytes == SOCKET_ERROR)
    return -1;
else
    return bytes;
```

NON-BLOCKING SOCKETS: ACCEPT()

1. Call as normal
2. Valid socket result = success
3. INVALID_SOCKET result = error
    1. If WSAEWOULDBLOCK, call again!

```
SOCKET result = accept(sock,
    incoming, &size);

if (result == INVALID_SOCKET)

    return -1;

return result;
```

NON-BLOCKING SOCKETS: CONNECT()

1. Call as normal
2. Non-error result = success
   1. But that won't happen
3. "Errors":
   1. WSAEWOULDBLOCK = try again!
   2. WSAEINVAL = ready to use!
   3. WSAEALREADY = ready to use!

```
int CheckConnect(SOCKET sock, sockaddr_in*
address)

{

    if (connect(sock, (sockaddr*)address,
        sizeof(sockaddr_in)) == SOCKET_ERROR)

    {

        int error = WSAGetLastError();
        if (error != WSAEINVAL && error !=
            WSAEALREADY)

            return 0;

    }

    return 1;

}
```

SELECT()

- int select(
  int nfds,
  fd_set* readfds,
  fd_set* writefds,
  fd_set* exceptfds,
  const struct timeval* timeout
  );

FD_ZERO(&fds);
FD_CLR(index, &fds);
FD_SET(index, &fds);
FD_ISSET(index, &fds);

- Still have to use non-blocking sockets, because Linux is naughty!
- fd_set is both input and output, so you have to reset it between calls.
- timeout is also both input and output on Linux, so you have to reset it between calls too.

SUMMARY

- Many techniques available—including just not bothering!—but the best ones in the context of this class are either multithreading, non-blocking sockets or select().
- Multithreading is hard. Don't do it unless you're doing it already.
- Non-blocking sockets are mostly easy.
- select() is only worthwhile if you need it for non-blocking stdio on POSIX.

Non-blocking sockets:

- Use ioctlsocket() to set FIONBIO flag to 1
- Treat WSAEWOULDBLOCK "error" as "try again"
- Connect() is a special case—success is represented by "errors" WSAEINVAL and/or WSAEALREADY