Non-blocking I/O

Warning: an unfamiliar and slightly non-intuitive topic...

Why did the sequential implementation do badly?

- it relied on **blocking** system calls
  - accept() blocked until a new connection arrived
  - read() blocked until new data arrived
  - write() potentially blocked until the write buffer had room
- nothing else could happen while the main thread blocks
Non-blocking I/O

An alternative: **non-blocking** network system calls

- non-blocking accept( )
  - if a connection is waiting, accept( ) succeeds and returns it
  - if no connection is waiting, accept( ) fails and returns immediately

- non-blocking read( )
  - if data is waiting, read( ) succeeds and returns it
  - if no data is waiting, read( ) fails and returns immediately

- non-blocking write( )
  - if buffer space is available, write( ) deposits data and returns
  - if no buffer space is available, write( ) fails and returns immediately
Reminder: threaded pseudocode

```c
// Start a thread for each connection
while (1) {
  fd = accept();
  pthread_create(t2, start, fd);
}

start(int fd) {
  while (1) {
    char *data = do_netread(fd);  // NET_READING
    do_netwrite(fd, data);        // NET_WRITING
  }
}

char *do_netread(int fd) {
  return read(fd);
}

void do_netwrite(int fd, char *data) {
  write(fd, data);
}
```
A (bad) attempt at non-blocking I/O

```c
state    s[N];            // clients’ state field
int     fd[N], readfd[N];  // clients’ file descriptors
char    *data[N], *fdata[N]; // buffers holding clients’ data

while (1) {
  if (fd = nb_accept())
    create state for new client, initialized to NET_READING;

  for (int i = 0; i < N; i++) {
    if (s[i] == NET_READING) {
      if (nb_read(fd[i], data[i]))
        s[i] = NET_WRITING;
    }

    if (s[i] == NET_WRITING) {
      if (nb_write(fd[i], fdata[i])
        s[i] = NET_READING;
    }
  }
}
```
Pictorially

\[\begin{array}{|c|c|c|}
\hline
\text{fd1} & \text{NET_WRITING} & \text{buffer} \\
\hline
\text{fd2} & \text{NET_READING} & \text{buffer} \\
\hline
\text{fd3} & \text{NET_READING} & \text{buffer} \\
\hline
\end{array}\]

main

\[
\text{buffer} = \text{nb_netread()} \\
\]

\[
\text{nb_accept()} \\
\]

\[
\text{do_netread()} \\
\text{do_netwrite()} \\
\]

\[
\text{while (1) { \\
    \text{accept( );} \\
    \text{thread_create(start);} \\
}}
\]

main

\[
\text{read()} \\
\]

\[
\text{write()} \\
\]

\[
\text{THREADED}
\]

\[
\text{NON BLOCKING}
\]
Task state
- kept in a table in the heap

Task concurrency, threads
- single thread dispatches
  “I/O is available” event
- program *is* task scheduler

Call graph
- only one “procedure” deep
- code path is **sliced** at what used to be blocking I/O
THREADED

Task state
- kept in each thread’s stack

Task concurrency, threads
- each thread spurts computation between long blocking IOs
- OS is the scheduler

Call graph
- many procedures deep; stack trace lines up with task progress

```
while (1) {
    accept();
    thread_create(start);
}
main
```
Problem with first attempt

It burns up the CPU, constantly looping
- testing each connection to see if it received an event
  ‣ if so, dispatch the event
- which events?
  ‣ `fd` is readable
  ‣ `fd` is writeable
  ‣ `fd` is acceptable
  ‣ `fd` closed / in an error state

```c
while (1) {
  if (fd = nb_accept())
    create state for new client, initialized to NET_READING;

  for (int i = 0; i < N; i++) {
    if (s[i] == NET_READING) {
      if (nb_read(fd[i], data[i]))
        s[i] = NET_WRITING;
    }
    if (s[i] == NET_WRITING) {
      if (nb_write(fd[i], fwrite[i]))
        s[i] = NET_READING;
    }
  }
}
```
An idea

Instead of constantly polling each file descriptor, why not have one blocking call?

- “hey OS, please tell me when the next event arrives”

```c
while (1) {
    (fd, event) = wait_for_next_event( fd_array );

    switch (event) {
       NET_ACCEPTABLE:
         (lookup_state, new_fd) = do_accept(fd);
         break;
       NET_WRITEABLE:
         do_netwrite(fd, lookup_state(fd));
         break;
       NET_READABLE:
         do_netread(fd, lookup_state(fd));
         break;
       NET_CLOSED:
         close(fd);
         break;
    }
}
```
select()

```c
int select(int nfds,
    fd_set *read_fds,
    fd_set *write_fds,
    fd_set *error_fds,
    struct timeval *timeout);
```

Waits (up to timeout) for one or more of the following:

- readable events on (read_fds)
- writeable events on (write_fds)
- error events on (error_fds)
see echo_concurrent_select.cc
See you on Wednesday!