CSE 333

Lecture 21 -- non-blocking I/O and select

Hal Perkins

Department of Computer Science & Engineering

University of Washington

Non-blocking I/O

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

Why do sequential server implementations do badly?

- they rely on **blocking** system calls
  - accept( ) blocks until a new connection arrived
  - read( ) blocks until new data arrived
  - write( ) potentially blocks until the write buffer had room
- nothing else can 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

```
// 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

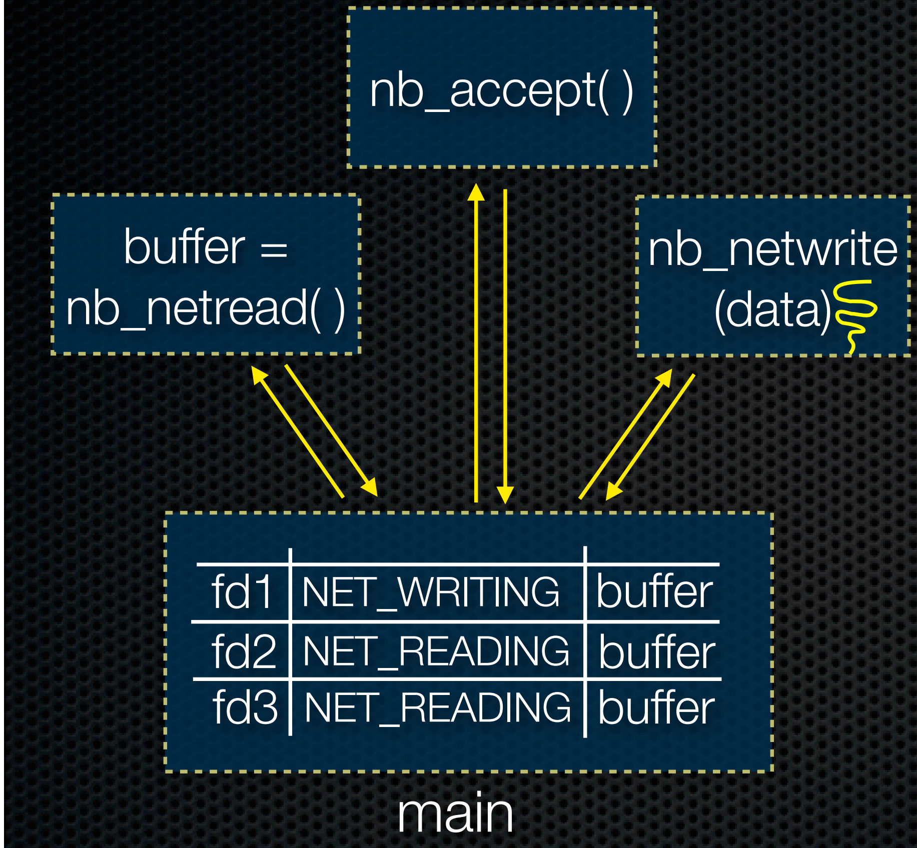
```
state    s[N];            // clients’ state field
int     fd[N], readfd[N]; // clients’ file descriptors
char *data[N], *fdata[N]; // buffers holding clients’ data

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



nb_accept( )

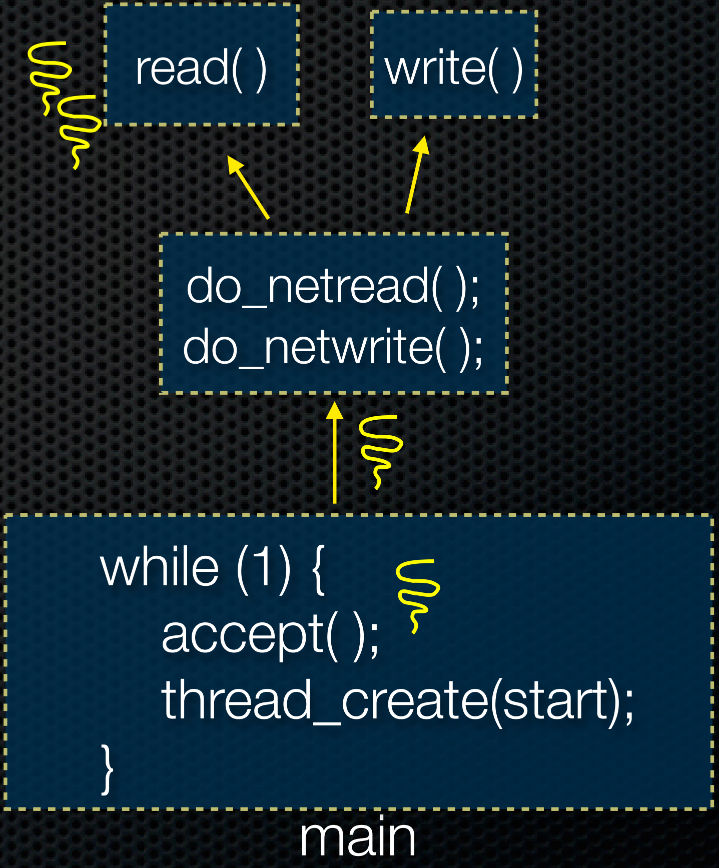
buffer = nb_netread( )

nb_netwrite (data)

fd1	NET_WRITING	buffer
fd2	NET_READING	buffer
fd3	NET_READING	buffer

main

NON BLOCKING



read( )

write( )

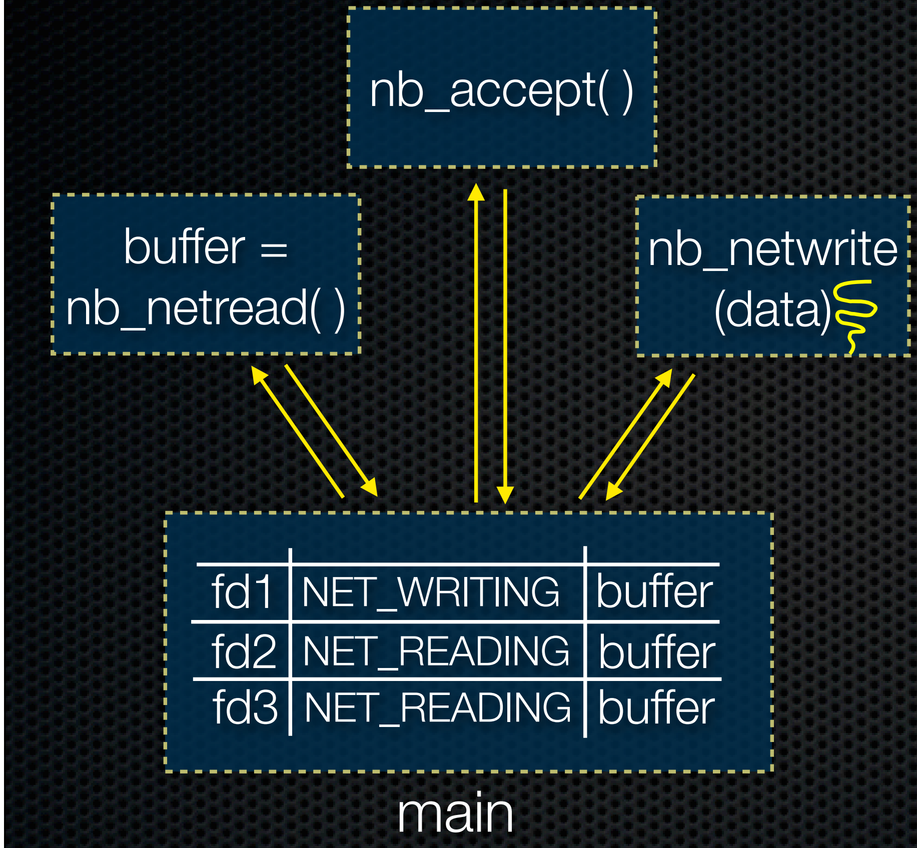
```
do_netread( );
do_netwrite( );
```

```
while (1) {
    accept( );
    thread_create(start);
}
```

main

THREADED

NON BLOCKING



nb_accept( )

buffer = nb_netread( )

nb_netwrite (data)

fd1	NET_WRITING	buffer
fd2	NET_READING	buffer
fd3	NET_READING	buffer

main

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

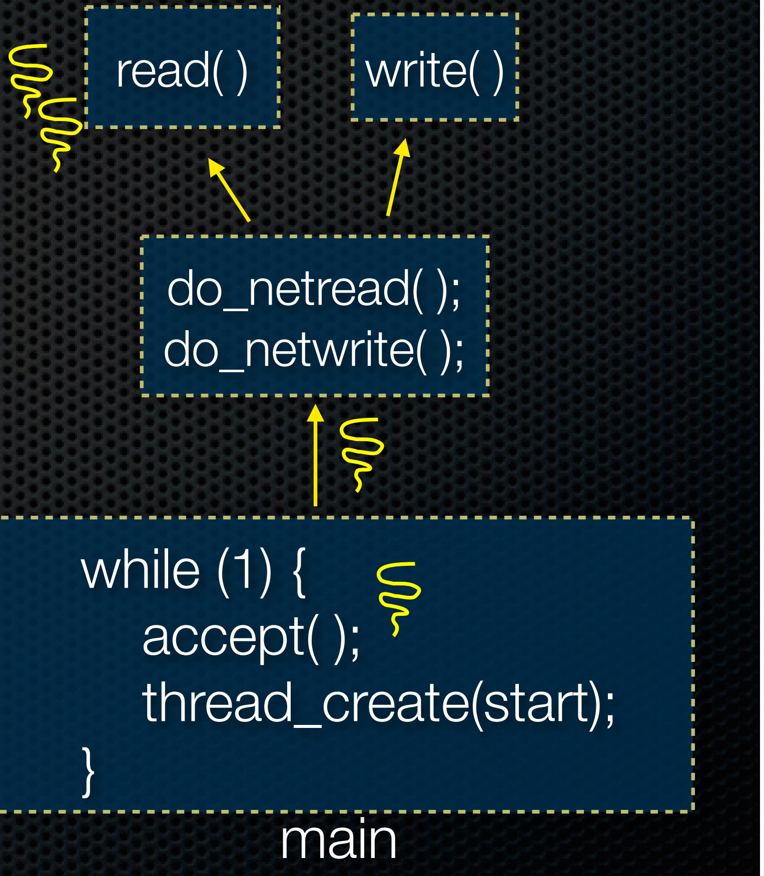
- kept in each thread’s stack

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);
}
```

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 read'able
  - fd is write'able
  - fd is accept'able
  - *fd closed / in an error state*

```
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;
    }
  }
}
```

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”

```
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( )

```
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

I/O Model Summary

Synchronous - requesting process waits until operation completes. Possible models:

- Blocking - operation completes only after I/O done; process suspended (if needed) until then.
- Non blocking - operation returns immediately, status indicates whether operation was done. Retry as needed.
- I/O multiplexing - use select to block until some operation completes. Block on select instead of actual I/O system call.
    - (Use with non-blocking I/O)

Asynchronous - requesting process not blocked

See you on Monday!