C Programmer's Tutorial: Mastering Non-Blocking Sockets with `select`

1. Introduction: Blocking vs. Non-Blocking I/O

Blocking Sockets (Default):

When you perform an operation (like `accept`, `connect`, `recv`, `send`) on a standard, blocking socket: * If the operation can be completed immediately (e.g., data is available to `recv`, buffer space is available for `send`, a connection is waiting for `accept`), it completes and returns. * If the operation *cannot* be completed immediately (e.g., no data to `recv`, send buffer is full, no incoming connection for `accept`, `connect` is in progress), the function *blocks*. This means your program's execution pauses at that line, waiting until the operation *can* complete.

This is simple to program initially but terrible for applications needing responsiveness or handling multiple clients concurrently without using threads or multiple processes for each connection. A single blocked call can freeze the entire application or prevent it from servicing other clients.

Non-Blocking Sockets:

When you set a socket to non-blocking mode: * If an operation can be completed immediately, it behaves like the blocking version (completes and returns). * If an operation *cannot* be completed immediately, the function returns *immediately* with an error status. It does *not* wait.

The key error indication for "operation would block" is the return value `-1`, with the global `errno` variable set to either `EWOULDBLOCK` or `EAGAIN`. On most modern systems (POSIX), these two constants have the same value and are interchangeable.

Why Use Non-Blocking Sockets?

- **Responsiveness:** Prevents a single I/O operation from freezing the entire application.
- **Scalability:** Allows a single thread/process to manage many network connections concurrently. Instead of waiting, the program can check which sockets are ready for I/O and service only those, effectively multiplexing I/O operations.
- **Fine-grained Control:** Gives the programmer explicit control over when and how to handle I/O readiness.

The challenge with non-blocking sockets is that you need a mechanism to know *when* an operation *would* succeed without blocking. This is where I/O multiplexing functions like `select`, `poll`, or `epoll`/`kqueue` come in. This tutorial focuses on the classic `select`.

2. Creating Non-Blocking Sockets: `fcntl` and `O_NONBLOCK`

Any socket descriptor (obtained from `socket()` or `accept()`) can be switched between blocking and non-blocking mode using the `fcntl()` (file control) system call.

Key Components:

- **`fcntl()`:** A versatile function to manipulate file descriptor properties.

  ```
  #include <fcntl.h>
  #include <unistd.h> // For STDIN_FILENO, STDOUT_FILENO if needed,
  though not socket specific

  int fcntl(int fd, int cmd, ... /* arg */ );
  ```

- **`fd`:** The file descriptor of the socket you want to modify.

- **`cmd`:** The command to execute. We need two:

  - `F_GETFL`: Get the current file status flags.
  - `F_SETFL`: Set the file status flags.
- **`arg` (for `F_SETFL`):** The new set of flags.

- **`O_NONBLOCK`:** The flag constant (defined in `<fcntl.h>`) that signifies non-blocking mode.

Procedure:

1. Get the current flags using `fcntl(fd, F_GETFL)`.
2. Check for errors after `F_GETFL`.
3. Modify the retrieved flags by adding `O_NONBLOCK` using the bitwise OR operator (`|`).
4. Set the modified flags using `fcntl(fd, F_SETFL, new_flags)`.
5. Check for errors after `F_SETFL`.

Example Function:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>
#include <sys/socket.h> // Needed for socket types, even if not
creating one here

// Function to set a socket descriptor to non-blocking mode
// Returns 0 on success, -1 on error (and sets errno)
int set_socket_non_blocking(int sockfd) {
    // Get current flags
    int flags = fcntl(sockfd, F_GETFL, 0);
    if (flags == -1) {
        perror("fcntl(F_GETFL)");
        return -1;
    }
```

```c
    // Add the O_NONBLOCK flag
    flags |= O_NONBLOCK;

    // Set the modified flags
    if (fcntl(sockfd, F_SETFL, flags) == -1) {
        perror("fcntl(F_SETFL)");
        return -1;
    }

    return 0; // Success
}

// Example Usage (assuming 'listener_fd' is a valid socket descriptor)
/*
if (set_socket_non_blocking(listener_fd) == -1) {
    // Handle error, maybe close the socket and exit
    close(listener_fd);
    exit(EXIT_FAILURE);
}
*/
```

Important: You typically set a socket to non-blocking *after* creating it with `socket()` and often *before* operations like `connect()` or starting the main `accept()` loop. For sockets returned by `accept()`, you must also set them to non-blocking individually if you want non-blocking behavior on the client connections.

3. Using Standard Socket Functions with Non-Blocking Sockets

Let's examine how common socket functions behave when the socket descriptor `sockfd` is in non-blocking mode.

- **`socket(domain, type, protocol)`:**
  - **Behavior:** Unchanged. Creates a socket endpoint. Returns a file descriptor or -1 on error.
  - **Non-Blocking Relevance:** The returned descriptor is initially *blocking* by default. You need to call `set_socket_non_blocking()` on it afterwards if desired.
- **`bind(sockfd, addr, addrlen)`:**
  - **Behavior:** Mostly unchanged. Assigns an address to the socket.
  - **Non-Blocking Relevance:** `bind` itself doesn't typically block for network reasons. Errors are usually immediate (e.g., address already in use (`EADDRINUSE`), invalid address (`EFAULT`), permissions (`EACCES`)). Non-blocking mode doesn't significantly alter its behavior. Returns 0 on success, -1 on error.
- **`listen(sockfd, backlog)`:**
  - **Behavior:** Unchanged. Marks the socket as passive (will accept incoming connections).

- **Non-Blocking Relevance:** `listen` is typically a quick, non-blocking operation itself. It sets up kernel queues. Non-blocking mode on `sockfd` doesn't affect `listen`. Returns 0 on success, -1 on error.

- **`accept(sockfd, addr, addrlen)`:**
  - **Blocking Behavior:** Waits indefinitely until a client connects. Returns a new socket descriptor for the connection.
  - **Non-Blocking Behavior:**
    - If one or more connections are pending in the queue, it dequeues the first one, creates a new socket descriptor for it, and returns that descriptor (a non-negative integer). The new descriptor *inherits the blocking/non-blocking mode* from the listening socket (`sockfd`) on some systems (like Linux), but POSIX doesn't guarantee this. **Best practice:** Explicitly set the desired mode (usually non-blocking) on the *newly accepted* socket descriptor using `fcntl()`.
    - If *no* connections are pending, it does *not* wait. It returns `-1` immediately, and `errno` is set to `EWOULDBLOCK` or `EAGAIN`.
  - **Error Handling:** Check the return value. If -1, check `errno`. If `errno` is `EWOULDBLOCK` or `EAGAIN`, it simply means "no connection waiting right now, try again later". Other `errno` values indicate genuine errors (e.g., `ECONNABORTED`, `EMFILE`, `ENFILE`, `ENOBUFS`, `EPERM`).

```
// Inside a loop after select() indicates listener_fd is readable
struct sockaddr_storage client_addr; // Use storage for IPv4/IPv6
compatibility
socklen_t client_addr_len = sizeof(client_addr);
int client_fd = accept(listener_fd, (struct sockaddr
*)&client_addr, &client_addr_len);

if (client_fd == -1) {
    if (errno == EWOULDBLOCK || errno == EAGAIN) {
        // This is expected, no connection waiting currently
        // Continue with other checks or loop again
    } else {
        // A real error occurred
        perror("accept");
        // Potentially handle specific errors or break the loop
    }
} else {
    // Successful accept!
    printf("Accepted connection on fd %d\n", client_fd);

    // *** CRITICAL: Set the new client socket to non-blocking
***
    if (set_socket_non_blocking(client_fd) == -1) {
        // Handle error, maybe close client_fd
        close(client_fd);
    } else {
        // Add client_fd to the set of descriptors managed by
```

```
select()
        // (More on this later)
    }
}
```

- **`connect(sockfd, addr, addrlen)`:**
  - **Blocking Behavior:** Initiates a connection. Waits (blocks) until the connection is established successfully or fails. Returns 0 on success, -1 on error.
  - **Non-Blocking Behavior:** This is one of the most complex.
    - It initiates the connection attempt *without waiting* for completion.
    - **Immediate Success:** If the connection can be established immediately (e.g., connecting to localhost), `connect` *might* return 0. This is relatively rare for TCP connections over a network.
    - **Immediate Failure:** If the system can determine failure immediately (e.g., network unreachable, connection refused locally), `connect` returns `-1`, and `errno` is set to the relevant error code (e.g., `ECONNREFUSED`, `ENETUNREACH`).
    - **Connection In Progress:** Most commonly for TCP, the connection establishment takes time (SYN -> SYN-ACK -> ACK). In this case, `connect` returns `-1` immediately, and `errno` is set to `EINPROGRESS`. This is *not* a fatal error; it means "the connection attempt has started, but hasn't finished yet."
  - **Handling `EINPROGRESS`:** When `connect` returns -1 with `errno == EINPROGRESS`, you need to:
    1. Wait for the socket to become *writable*. This indicates that the TCP handshake has either completed successfully or failed. Use `select()` (or `poll`/`epoll`) to monitor the socket descriptor (`sockfd`) for writability.

    2. Once `select` reports the socket as writable, you *must* check the actual outcome of the connection attempt. Use `getsockopt()` to retrieve the socket-level error status.

       ```
       int error = 0;
       socklen_t len = sizeof(error);
       if (getsockopt(sockfd, SOL_SOCKET, SO_ERROR, &error,
       &len) < 0) {
           // Error checking getsockopt itself (rare)
           perror("getsockopt SO_ERROR");
           // Connection status is unknown, probably best to
       close sockfd
           close(sockfd);
           // Handle error...
       } else {
           if (error == 0) {
               // Success! Connection established.
       ```

```c
        printf("Connection successful on fd %d\n",
sockfd);
        // Socket is now ready for send/recv
    } else {
        // Failure! The connection attempt failed.
        fprintf(stderr, "Connection failed on fd %d:
%s\n", sockfd, strerror(error));
        // Close the socket
        close(sockfd);
        // Handle error...
    }
}
```

- The `error` variable retrieved by `getsockopt` will contain the actual result: 0 for success, or an `errno` code (like `ECONNREFUSED`, `ETIMEDOUT`) indicating the reason for failure.

- **`send(sockfd, buf, len, flags)` / `sendto(sockfd, buf, len, flags, dest_addr, addrlen)`:**
  - **Blocking Behavior:** Waits until *some* data can be sent into the socket's send buffer. May not send all `len` bytes if the buffer is smaller. Returns the number of bytes actually sent, or -1 on error.
  - **Non-Blocking Behavior:**
    - If there is space in the send buffer, it copies as much data as possible (up to `len` bytes) into the buffer *immediately* and returns the number of bytes copied. This might be less than `len` (a "partial send").
    - If the send buffer is completely full, it does *not* wait. It returns `-1` immediately, and `errno` is set to `EWOULDBLOCK` or `EAGAIN`.
  - **Handling Partial Sends:** Because `send`/`sendto` might return a value less than `len`, you *must* loop if you need to guarantee all data is sent. Keep track of how many bytes have been sent and adjust the buffer pointer and remaining length for the next call. Only stop looping when all bytes are sent or an error other than `EWOULDBLOCK`/`EAGAIN` occurs. If you get `EWOULDBLOCK`/`EAGAIN`, you need to wait (using `select`) for the socket to become *writable* again before retrying the `send`.

```c
const char *buffer = "Data to send";
size_t total_to_send = strlen(buffer);
size_t total_sent = 0;

while (total_sent < total_to_send) {
    ssize_t sent_now = send(sockfd, buffer + total_sent,
total_to_send - total_sent, 0); // Use MSG_NOSIGNAL often too

    if (sent_now == -1) {
        if (errno == EWOULDBLOCK || errno == EAGAIN) {
            // Buffer is full, need to wait for writability
            printf("Send buffer full on fd %d, will retry later.\
n", sockfd);
```

```c
            // Use select() to wait for writability before trying again
            // Break out of this inner send loop for now
            break;
        } else {
            // A real error occurred
            perror("send");
            // Handle error (e.g., close socket)
            close(sockfd);
            // Indicate failure
            total_sent = -1; // Or some other error flag
            break;
        }
    } else if (sent_now == 0) {
         // send() returning 0 is unusual but possible in some edge cases.
         // Treat as potentially problematic or buffer full.
         fprintf(stderr, "send() returned 0 on fd %d. Assuming buffer full.\n", sockfd);
         // Use select() to wait for writability.
         break;
    } else {
        // Successfully sent 'sent_now' bytes
        total_sent += sent_now;
        printf("Sent %zd bytes on fd %d (total %zu/%zu)\n", sent_now, sockfd, total_sent, total_to_send);
    }
}

// After the loop:
// If total_sent == total_to_send, all data is buffered.
// If total_sent < total_to_send (and not -1), need to retry sending the rest later.
// If total_sent == -1 (or error flag), an error occurred.
```

  - **EPIPE Error:** If you try to `send` on a socket where the remote end has closed the connection for reading (sent `FIN` and you received it, or connection reset), you might get an `EPIPE` error. This usually also raises a `SIGPIPE` signal. It's often desirable to ignore `SIGPIPE` (using `signal(SIGPIPE, SIG_IGN);`) and handle the `EPIPE` error return from `send` directly, typically by closing the socket. Using the `MSG_NOSIGNAL` flag in the `send` call (if available on your system) prevents the signal and just returns `EPIPE`.
- **`recv(sockfd, buf, len, flags)` / `recvfrom(sockfd, buf, len, flags, src_addr, addrlen)`:**
  - **Blocking Behavior:** Waits until *some* data arrives in the socket's receive buffer. Returns the number of bytes read (up to `len`), 0 if the peer closed the connection gracefully, or -1 on error.
  - **Non-Blocking Behavior:**

        - If data is available in the receive buffer, it reads as much as possible (up to `len` bytes) *immediately* and returns the number of bytes read. This could be less than `len`.
        - If *no* data is available in the receive buffer, it does *not* wait. It returns `-1` immediately, and `errno` is set to `EWOULDBLOCK` or `EAGAIN`.
        - If the peer has performed an orderly shutdown (sent a `FIN`), `recv` returns `0`. This indicates the end of the data stream from the peer. You should typically close your end of the connection too after reading any remaining buffered data.
        - If a connection reset occurs (peer sent `RST`), `recv` returns `-1`, and `errno` is set to `ECONNRESET`.
    - **Handling `EWOULDBLOCK`/`EAGAIN`:** If you get this error, it means "no data available right now, try again later". You need to wait (using `select`) for the socket to become *readable* again before retrying the `recv`.
    - **Handling Return Value 0:** This is crucial. It signifies a graceful shutdown by the peer. You should stop trying to read from this socket and usually close it.
    - **Handling Partial Reads:** Like `send`, `recv` might not fill your entire buffer (`len` bytes) even if more data is coming. If your application protocol expects messages of a specific size, you may need to loop, calling `recv` multiple times and accumulating data until you have a complete message or encounter an error or EOF (return 0).

```
char buffer[1024];
ssize_t bytes_received;

// Inside a loop after select() indicates sockfd is readable
bytes_received = recv(sockfd, buffer, sizeof(buffer) - 1, 0); // Leave space for null terminator if needed

if (bytes_received == -1) {
    if (errno == EWOULDBLOCK || errno == EAGAIN) {
        // No data available right now, expected in non-blocking mode
        // Continue checking other sockets or loop again
    } else if (errno == ECONNRESET) {
         // Connection reset by peer
         fprintf(stderr, "Connection reset by peer on fd %d\n", sockfd);
         close(sockfd);
         // Remove sockfd from select() management
    } else {
        // A real error occurred
        perror("recv");
        close(sockfd);
        // Remove sockfd from select() management
    }
} else if (bytes_received == 0) {
    // Peer closed the connection gracefully (EOF)
```

```c
        printf("Connection closed by peer on fd %d\n", sockfd);
        close(sockfd);
        // Remove sockfd from select() management
    } else {
        // Successfully received 'bytes_received' bytes
        buffer[bytes_received] = '\0'; // Null-terminate if treating
as string
        printf("Received %zd bytes on fd %d: %s\n", bytes_received,
sockfd, buffer);
        // Process the received data...
        // Note: Might need to accumulate data if expecting larger
messages
    }
```

4. Checking Socket Readiness: The `select()` System Call

Since non-blocking calls return immediately if they can't complete, we need a way to know *when* to call them. We don't want to "busy-wait" (calling `recv` in a tight loop until it stops returning `EWOULDBLOCK`), as that wastes CPU.

`select()` allows a program to monitor multiple file descriptors, waiting until one or more become "ready" for a certain class of I/O operation (input, output, or exceptional conditions).

Key Components:

- **`select()` function:**

  ```c
  #include <sys/select.h> // Or <sys/time.h>, <sys/types.h>,
  <unistd.h> depending on system

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

- **`fd_set`:** An opaque data type representing a set of file descriptors. Think of it as a bitmask or array, but use the provided macros to manipulate it.

- **Macros for `fd_set`:**

  - `FD_ZERO(fd_set *set)`: Clears the set (initializes it to contain no file descriptors). **Must be called before using a set in `select` each time.**
  - `FD_SET(int fd, fd_set *set)`: Adds file descriptor `fd` to the set.
  - `FD_CLR(int fd, fd_set *set)`: Removes file descriptor `fd` from the set.
  - `FD_ISSET(int fd, fd_set *set)`: Returns non-zero (true) if `fd` is present in the set, zero (false) otherwise. Used *after* `select` returns to check which descriptors are ready.
- **`nfds`:** The highest-numbered file descriptor in any of the three sets, *plus 1*. `select` only checks descriptors from 0 up to `nfds - 1`. Calculating this correctly is crucial for performance and correctness.

- **`readfds`:** Pointer to an `fd_set`. `select` will wait until one of the descriptors in this set is ready for reading. Ready for reading means:
  - Data is available to be read (`recv`, `recvfrom`).
  - For a listening socket, an incoming connection is pending (`accept`).
  - End-of-file (graceful shutdown) has been received.
  - An error condition is pending (reading will return an error like `ECONNRESET`).
- **`writefds`:** Pointer to an `fd_set`. `select` will wait until one of the descriptors in this set is ready for writing. Ready for writing usually means:
  - There is space available in the socket send buffer (`send`, `sendto`). (Note: Even if ready, a subsequent `send` might still only perform a partial write or even block if the buffer fills up quickly).
  - For a non-blocking `connect`, writability indicates the connection attempt has completed (either successfully or with an error). You *must* then use `getsockopt(SO_ERROR)` to check the outcome.
  - An error condition is pending (writing will return an error like `EPIPE`).
- **`exceptfds`:** Pointer to an `fd_set`. Monitors for "exceptional conditions". For TCP sockets, this typically means the arrival of Out-Of-Band (OOB) data. Its use is less common than `readfds` and `writefds`.
- **`timeout`:** Pointer to a `struct timeval` specifying the maximum time `select` should wait.
  - `struct timeval { time_t tv_sec; suseconds_t tv_usec; };`
  - If `timeout` is `NULL`: `select` blocks indefinitely until a descriptor is ready.
  - If `timeout` points to a struct with `tv_sec = 0` and `tv_usec = 0`: `select` returns immediately after checking the descriptors (polling).
  - If `timeout` points to a struct with positive values: `select` waits up to the specified time.
  - **Important:** Some older Unix systems (and potentially Linux, though often fixed now) *modify* the `timeout` struct to reflect the time remaining. POSIX allows this. For portability, if you need a fixed timeout in a loop, re-initialize the `struct timeval` before *each* call to `select`.

Return Value of `select()`:

- **> 0:** The number of file descriptors that are ready across all the sets. The sets (`readfds`, `writefds`, `exceptfds`) are *modified* in place to indicate *which* descriptors are ready. You must use `FD_ISSET` to check each descriptor you were interested in.
- **0:** The timeout expired before any descriptors became ready.
- **-1:** An error occurred. `errno` is set (e.g., `EBADF` for an invalid descriptor in one of the sets, `EINTR` if interrupted by a signal, `EINVAL` for invalid `nfds` or timeout).

Typical `select()` Loop Structure (Server Example):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/select.h>
#include <errno.h>
#include <fcntl.h> // For O_NONBLOCK
#include <signal.h> // For signal handling (e.g., SIGPIPE)

#define PORT 8080
#define MAX_CLIENTS 30
#define BUFFER_SIZE 1024

// (Include set_socket_non_blocking function from earlier)
int set_socket_non_blocking(int sockfd) { /* ... as above ... */ }

int main() {
    int listener_fd, new_fd;
    struct sockaddr_in server_addr, client_addr;
    socklen_t addr_len;
    char buffer[BUFFER_SIZE];

    int client_sockets[MAX_CLIENTS] = {0}; // Array to hold client
socket fds
    int max_clients = MAX_CLIENTS;
    int activity, i, valread, sd;
    int max_sd; // Max descriptor value needed for select()

    fd_set readfds; // Set of socket descriptors for select()

    // Ignore SIGPIPE or handle it appropriately
    signal(SIGPIPE, SIG_IGN);

    // 1. Create listening socket
    if ((listener_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    // Optional: Allow reuse of address (useful for quick restarts)
    int opt = 1;
    if (setsockopt(listener_fd, SOL_SOCKET, SO_REUSEADDR, (char
*)&opt, sizeof(opt)) < 0) {
        perror("setsockopt SO_REUSEADDR");
        // Non-fatal, continue
```

```
    }

    // 2. Set listener socket to non-blocking
    if (set_socket_non_blocking(listener_fd) == -1) {
        close(listener_fd);
        exit(EXIT_FAILURE);
    }

    // 3. Bind
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY; // Listen on all
interfaces
    server_addr.sin_port = htons(PORT);

    if (bind(listener_fd, (struct sockaddr *)&server_addr,
sizeof(server_addr)) < 0) {
        perror("bind failed");
        close(listener_fd);
        exit(EXIT_FAILURE);
    }
    printf("Listener bound to port %d\n", PORT);

    // 4. Listen
    if (listen(listener_fd, 5) < 0) { // Backlog of 5 pending
connections
        perror("listen failed");
        close(listener_fd);
        exit(EXIT_FAILURE);
    }
    printf("Waiting for connections ...\n");

    // 5. Main Server Loop
    while (1) {
        // --- Prepare fd_set for select() ---
        FD_ZERO(&readfds);

        // Add listener socket to the set
        FD_SET(listener_fd, &readfds);
        max_sd = listener_fd;

        // Add child sockets (clients) to the set
        for (i = 0; i < max_clients; i++) {
            sd = client_sockets[i];

            // If valid socket descriptor then add to read list
            if (sd > 0) {
                FD_SET(sd, &readfds);
            }
```

```
            // Update max_sd if needed (for the nfds parameter)
            if (sd > max_sd) {
                max_sd = sd;
            }
        }

        // --- Wait for activity on any socket ---
        // Timeout is NULL = wait indefinitely
        // Can use a timeval struct for timeouts
        // struct timeval tv;
        // tv.tv_sec = 5; // 5 second timeout
        // tv.tv_usec = 0;
        // activity = select(max_sd + 1, &readfds, NULL, NULL, &tv);

        activity = select(max_sd + 1, &readfds, NULL, NULL, NULL);

        if ((activity < 0) && (errno != EINTR)) {
            // EINTR means select was interrupted by a signal, safe to
continue
            perror("select error");
            // Consider more robust error handling here
            break; // Exit loop on critical select error
        }
        if (activity == 0) {
            // Timeout occurred (if timeout was set)
            printf("select() timed out.\n");
            continue;
        }

        // --- Service Ready Sockets ---

        // A. Check listener socket for incoming connection
        if (FD_ISSET(listener_fd, &readfds)) {
            addr_len = sizeof(client_addr);
            new_fd = accept(listener_fd, (struct sockaddr
*)&client_addr, &addr_len);

            if (new_fd == -1) {
                if (errno == EWOULDBLOCK || errno == EAGAIN) {
                    // This shouldn't happen if select indicated
readability,
                    // but handle defensively. Could occur in rare
edge cases
                    // (e.g., connection reset before accept).
                    fprintf(stderr, "accept returned
EWOULDBLOCK/EAGAIN despite select\n");
                } else {
                    perror("accept failed");
                    // Potentially serious error on listener
                }
```

```
            } else {
                // Connection accepted
                printf("New connection: socket fd %d, IP: %s, Port:
%d\n",
                       new_fd, inet_ntoa(client_addr.sin_addr),
ntohs(client_addr.sin_port));

                // Set the new socket to non-blocking
                if (set_socket_non_blocking(new_fd) == -1) {
                    close(new_fd);
                } else {
                    // Add new socket to array of sockets
                    for (i = 0; i < max_clients; i++) {
                        if (client_sockets[i] == 0) {
                            client_sockets[i] = new_fd;
                            printf("Adding client fd %d to index %d\
n", new_fd, i);
                            break;
                        }
                    }
                    if (i == max_clients) {
                         fprintf(stderr, "Too many clients, rejecting
fd %d\n", new_fd);
                         close(new_fd);
                    }
                }
            }
        } // End check listener_fd

        // B. Check client sockets for incoming data
        for (i = 0; i < max_clients; i++) {
            sd = client_sockets[i];

            if (sd > 0 && FD_ISSET(sd, &readfds)) {
                // Socket 'sd' is ready for reading
                memset(buffer, 0, BUFFER_SIZE); // Clear buffer
                valread = recv(sd, buffer, BUFFER_SIZE - 1, 0);

                if (valread == -1) {
                    if (errno == EWOULDBLOCK || errno == EAGAIN) {
                        // Should not happen often if select reported
readable,
                        // but possible if data was consumed elsewhere
or edge case.
                        fprintf(stderr, "recv EWOULDBLOCK/EAGAIN on fd
%d despite select\n", sd);
                    } else if (errno == ECONNRESET) {
                        // Connection reset by client
                        printf("Client fd %d disconnected
(ECONNRESET)\n", sd);
```

```
                        close(sd);
                        client_sockets[i] = 0; // Mark as free
                    } else {
                        // Other recv error
                        perror("recv failed");
                        close(sd);
                        client_sockets[i] = 0; // Mark as free
                    }
                } else if (valread == 0) {
                    // Client disconnected gracefully (EOF)
                    getpeername(sd, (struct sockaddr*)&client_addr,
&addr_len);
                    printf("Client fd %d disconnected gracefully, IP
%s, Port %d\n",
                           sd, inet_ntoa(client_addr.sin_addr),
ntohs(client_addr.sin_port));

                    // Close the socket and mark as 0 in array for
reuse
                    close(sd);
                    client_sockets[i] = 0;
                } else {
                    // Data received successfully
                    buffer[valread] = '\0'; // Null-terminate
                    printf("Received %d bytes from client fd %d: %s\
n", valread, sd, buffer);

                    // Example: Echo data back to client
                    // Note: This send should ideally also be non-
blocking and
                    // handled with writefds in select() if it might
block.
                    // For simplicity here, we do a blocking send or
simple non-blocking try.
                    ssize_t sent_bytes = send(sd, buffer, valread, 0);
// Use MSG_NOSIGNAL if available
                    if (sent_bytes == -1) {
                         if (errno == EWOULDBLOCK || errno == EAGAIN)
{
                              fprintf(stderr, "Send buffer full on fd
%d, data lost (in this simple example)\n", sd);
                              // A real app would buffer this data and
use writefds
                         } else if (errno == EPIPE) {
                              printf("Client fd %d disconnected (EPIPE
on send)\n", sd);
                              close(sd);
                              client_sockets[i] = 0;
                         } else {
                              perror("send failed");
```

```c
                                close(sd);
                                client_sockets[i] = 0;
                            }
                        } else if (sent_bytes < valread) {
                            fprintf(stderr, "Partial send on fd %d
(%zd/%d bytes), data lost (in this simple example)\n", sd, sent_bytes,
valread);
                            // A real app would buffer remaining data and
use writefds
                        }
                    }
                } // End check FD_ISSET(sd, &readfds)
            } // End loop through client sockets
    } // End main while(1) loop

    // Cleanup (won't be reached in this infinite loop example)
    printf("Shutting down server.\n");
    for (i = 0; i < max_clients; i++) {
        if (client_sockets[i] > 0) {
            close(client_sockets[i]);
        }
    }
    close(listener_fd);

    return 0;
}
```

Key Points about the `select` Loop:

1. **Reinitialize `fd_sets`:** `FD_ZERO` and `FD_SET` must be called *inside* the loop before each `select` call because `select` modifies the sets.
2. **Calculate `nfds`:** Keep track of the highest descriptor value (`max_sd`) and pass `max_sd + 1` to `select`.
3. **Check `select` Return:** Handle errors (`-1`), timeouts (`0`), and readiness (`> 0`).
4. **Use `FD_ISSET`:** After `select` returns `> 0`, iterate through *all* the descriptors you put into the sets and use `FD_ISSET` to find out *which specific ones* are ready.
5. **Handle Readiness:** Perform the appropriate non-blocking operation (`accept`, `recv`, `send`, `getsockopt` for `connect`).
6. **Handle Non-Blocking Errors:** Be prepared to handle `EWOULDBLOCK`/`EAGAIN` even after `select` indicates readiness, although it should be less frequent. Handle `0` return from `recv` (EOF) and `ECONNRESET`.
7. **Manage Client State:** Keep track of connected clients (e.g., in an array or list). Add new clients after `accept`, remove them on disconnect/error. Update `max_sd` accordingly.
8. **Write Readiness:** The example above only uses `readfds`. A full application often needs `writefds` too:
   - To know when a non-blocking `connect` has finished.

- To know when you can resume sending data after a previous `send` returned `EWOULDBLOCK`/`EAGAIN`. You would only add a client socket to `writefds` if you have pending data to send to it. Once the data is sent (or buffered successfully), remove it from `writefds` for the next `select` iteration to avoid busy-waiting when there's nothing to write.

5. Handling Non-Blocking `connect()` in Detail

As mentioned, `connect` returning `-1` with `errno == EINPROGRESS` requires special handling using `select`.

Steps:

1. Create the socket (`socket`).
2. Set it to non-blocking (`set_socket_non_blocking`).
3. Call `connect`.
4. **Check `connect` return:**
   - If `0`: Success (rare). Socket is ready.
   - If `-1` and `errno != EINPROGRESS`: Immediate failure. Close socket, report error.
   - If `-1` and `errno == EINPROGRESS`: Connection attempt started. Proceed to step 5.
5. **Wait for Writability:**
   - Add the socket descriptor `sockfd` to the `writefds` set for `select`.
   - Optionally, also add it to `readfds` (sometimes errors are reported via readability). It's generally safer to check `SO_ERROR` after writability.
   - Call `select` with an appropriate timeout.
6. **Check `select` return:**
   - If `0`: Timeout occurred before connection completed. You might retry `select`, or abort and close the socket.
   - If `-1`: `select` error. Handle it.
   - If `> 0`: Check if `sockfd` is set in `writefds` (or `readfds` if you added it there).
7. **Verify Connection Outcome:** If `FD_ISSET(sockfd, &writefds)` is true:
   - Use `getsockopt(sockfd, SOL_SOCKET, SO_ERROR, ...)` to retrieve the pending error status.
   - Check the retrieved error value:
     - If `0`: Connection successful! The socket is now connected and ready for `send`/`recv`. Remove it from the `writefds` check (unless you immediately have data to send).
     - If non-zero (e.g., `ECONNREFUSED`, `ETIMEDOUT`): Connection failed. Close the socket, report the specific error.

Example Snippet (Conceptual):

```c
// Assume sockfd created and set non-blocking
int result = connect(sockfd, (struct sockaddr*)&server_addr,
sizeof(server_addr));

if (result == 0) {
    printf("Connection immediate success (rare).\n");
    // Socket is ready
} else if (result == -1 && errno == EINPROGRESS) {
    printf("Connection in progress...\n");

    fd_set writefds;
    struct timeval tv;
    int select_ret;
    int sock_err = 0;
    socklen_t err_len = sizeof(sock_err);

    FD_ZERO(&writefds);
    FD_SET(sockfd, &writefds);

    // Set a timeout (e.g., 10 seconds)
    tv.tv_sec = 10;
    tv.tv_usec = 0;

    select_ret = select(sockfd + 1, NULL, &writefds, NULL, &tv);

    if (select_ret == -1) {
        perror("select during connect");
        close(sockfd);
        // Handle error
    } else if (select_ret == 0) {
        fprintf(stderr, "Connect timeout.\n");
        close(sockfd);
        // Handle timeout
    } else {
        // select_ret > 0, check if our socket is writable
        if (FD_ISSET(sockfd, &writefds)) {
            // Socket is writable, check SO_ERROR
            if (getsockopt(sockfd, SOL_SOCKET, SO_ERROR, &sock_err,
&err_len) == -1) {
                perror("getsockopt SO_ERROR");
                close(sockfd);
                // Handle error
            } else {
                if (sock_err == 0) {
                    printf("Connection successful!\n");
                    // Socket is ready for I/O
                } else {
                    // Connection failed, sock_err contains the errno
                    fprintf(stderr, "Connection failed: %s\n",
strerror(sock_err));
```

```
                    close(sockfd);
                    // Handle error
                }
            }
        } else {
            // This shouldn't happen if select returned > 0 for only
this fd
            fprintf(stderr, "select reported ready, but FD_ISSET is
false?\n");
            close(sockfd);
        }
    }
} else {
    // Immediate connect error
    perror("connect failed immediately");
    close(sockfd);
    // Handle error
}
```

6. Non-Blocking Raw Sockets

Raw sockets (`SOCK_RAW`) allow you to send/receive network packets directly at the IP layer (or sometimes even lower), bypassing much of the kernel's protocol processing (like TCP/UDP headers, checksums, fragmentation/reassembly for TCP).

Using Non-Blocking Mode with Raw Sockets:

The principles of non-blocking I/O apply to raw sockets just as they do to TCP or UDP sockets:

1. **Creation:** Create the raw socket using `socket(domain, SOCK_RAW, protocol)`. Common domains are `AF_INET` (IPv4) and `AF_INET6` (IPv6). The `protocol` argument specifies which IP protocol number you want to handle (e.g., `IPPROTO_ICMP`, `IPPROTO_UDP`, `IPPROTO_TCP`, or `IPPROTO_RAW` for crafting full IP headers).

   ```
   // Example: Raw socket to receive all IPv4 protocols
   // Needs CAP_NET_RAW capability or root privileges
   int raw_sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
   if (raw_sock < 0) {
       perror("socket(SOCK_RAW)");
       // exit or handle error
   }
   ```

2. **Setting Non-Blocking:** Use `fcntl(raw_sock, F_SETFL, O_NONBLOCK)` exactly as shown before.

3. **Sending (`sendto`):**

- You typically construct the *entire* IP packet, including the IP header (unless you used a protocol like `IPPROTO_ICMP` where the kernel might help, or if the `IP_HDRINCL` socket option is *not* set).
- Use `sendto` to specify the destination IP address.
- **Non-Blocking Behavior:** If the network interface's output queue or internal kernel buffers are full, `sendto` on a non-blocking raw socket will return `-1` with `errno` set to `EWOULDBLOCK/EAGAIN`.
- You would use `select` monitoring `writefds` to know when you can try sending again.

4. **Receiving (`recvfrom`):**

   - You will receive incoming IP packets that match the protocol specified when creating the socket (or all protocols if `IPPROTO_RAW` was used).
   - The buffer you provide to `recvfrom` will contain the *full* packet, including the IP header and payload. You need to parse these headers yourself.
   - `recvfrom` will also fill in the source address structure.
   - **Non-Blocking Behavior:** If no matching packets have arrived, `recvfrom` on a non-blocking raw socket returns `-1` with `errno` set to `EWOULDBLOCK/EAGAIN`.
   - You use `select` monitoring `readfds` to know when a packet is available to be read.

5. **`select` Usage:** Use `select` with `readfds` (to check for incoming packets) and `writefds` (to check if you can send) exactly as with other socket types.

Key Differences/Considerations for Raw Sockets:

- **Privileges:** Creating and using raw sockets typically requires elevated privileges (`CAP_NET_RAW` capability on Linux, or root access).
- **Header Handling:** You are responsible for constructing/parsing IP headers (and potentially transport layer headers depending on the protocol and `IP_HDRINCL` option). This involves understanding IP header fields, checksum calculation, etc.
- **No Connection State (Mostly):** Raw sockets are inherently connectionless. `connect` can sometimes be used on raw sockets to set a default destination address (so you can use `send/recv`), but it doesn't establish a "connection" in the TCP sense. `accept` and `listen` are not used.
- **Error Handling:** Errors like `EWOULDBLOCK/EAGAIN` work the same, but other errors might relate more directly to IP-level issues or permissions.

Essentially, the *non-blocking mechanism* (`fcntl`, `select`, handling `EWOULDBLOCK/EAGAIN`) is identical. The complexity comes from the *raw* nature requiring manual packet manipulation and higher privileges.

7. Corner Cases and Common Errors

- **`EWOULDBLOCK` vs. `EAGAIN`:** POSIX allows these to be different, but on most modern systems (Linux, BSDs), they are the same value. Write your code to check for both: `if (errno == EWOULDBLOCK || errno == EAGAIN)`.
- **`EINTR`:** System calls (like `select`, `recv`, `send`, `accept`, `connect`) can be interrupted by signals. If a call returns `-1` and `errno` is `EINTR`, it doesn't mean an error occurred with the socket itself. The standard practice is to simply retry the call immediately. Your main `select` loop should check for this: `if ((activity < 0) && (errno != EINTR)) { perror("select"); ... }`.
- **Spurious Wakeups:** `select` might occasionally return indicating a descriptor is ready, but the subsequent non-blocking call (e.g., `recv`) returns `EWOULDBLOCK/EAGAIN`. This can happen due to race conditions or specific kernel behaviors. Your code *must* handle `EWOULDBLOCK/EAGAIN` gracefully even after `select` indicated readiness. Don't assume the operation will succeed without blocking.
- **`select` Modifies `fd_sets` and `timeout`:** Always re-initialize `fd_sets` (`FD_ZERO`, `FD_SET`) and the `timeout` struct (if used) before each call to `select` in a loop.
- **`nfds` Calculation:** Forgetting the `+ 1` in `select(max_sd + 1, ...)` is a common error, leading `select` to ignore the highest-numbered descriptor. Always track the maximum `fd` and add one.
- **Forgetting `O_NONBLOCK` on Accepted Sockets:** A listening socket might be non-blocking, but the socket returned by `accept` might be blocking by default (behavior varies). Always explicitly set the accepted socket to non-blocking if that's the desired mode for client connections.
- **Handling `send`/`recv` Return Values:**
  - Always check for `-1`.
  - If `-1`, check `errno` (`EWOULDBLOCK/EAGAIN`, `EINTR`, `ECONNRESET`, `EPIPE`, etc.).
  - For `recv`, check for `0` (EOF).
  - For `send`/`recv`, handle partial reads/writes (return value > 0 but less than requested). Buffer data and retry later using `select` to check readiness again.
- **`getsockopt(SO_ERROR)` after `connect`:** Forgetting to check `SO_ERROR` after `select` indicates writability for a non-blocking `connect` means you don't know if the connection succeeded or failed.
- **Resource Limits (`EMFILE`, `ENFILE`):** `accept` or `socket` can fail with `EMFILE` (per-process limit on file descriptors reached) or `ENFILE` (system-wide limit reached). Gracefully handle this, perhaps by temporarily stopping accepting connections or logging the error. Check your system's limits (`ulimit -n`).
- **Buffer Management:** Non-blocking I/O often requires explicit buffer management. If `send` returns `EWOULDBLOCK`, you need to store the unsent data somewhere until `select` indicates the socket is writable again. If `recv` returns only part of a message, you need to buffer it and wait for the rest.

- **Closing Descriptors:** Ensure you `close()` sockets when they are no longer needed (EOF, error, server shutdown). Failing to do so leaks file descriptors. Also, remove closed descriptors from your active set managed by `select`. Forgetting to remove them can lead to `select` reporting readiness on invalid descriptors, causing `EBADF` errors on subsequent I/O calls.
- **`SIGPIPE`:** As mentioned, writing to a socket whose read end has been closed can trigger `SIGPIPE`, terminating the process by default. Either ignore the signal (`signal(SIGPIPE, SIG_IGN);`) or use the `MSG_NOSIGNAL` flag with `send`/`sendto` (if available) and handle the `EPIPE` error return instead.

8. Alternatives to `select`

While `select` is classic and portable, it has limitations: * **`FD_SETSIZE` Limit:** The maximum number of descriptors `select` can handle is often limited by the `FD_SETSIZE` constant (traditionally 1024, though sometimes higher). This limits scalability for servers with thousands of connections. * **Performance:** `select` modifies the `fd_set`s, requiring the kernel to check every descriptor in the sets up to `nfds` on each call, and requiring the user program to re-build the sets and iterate through them after `select` returns. This becomes inefficient with many descriptors.

Modern alternatives offer better performance and scalability: * **`poll()`:** Similar to `select` but uses an array of `struct pollfd` instead of `fd_set`s. Doesn't have the hard `FD_SETSIZE` limit and doesn't modify the input array, but still requires iterating. More portable than `epoll`/`kqueue`. * **`epoll` (Linux):** An event-based mechanism. You register descriptors with an `epoll` instance once. The kernel then tells you *only* which descriptors are ready. Much more efficient for large numbers of descriptors, especially sparse ones (where only a few are active at any time). Offers edge-triggered (ET) and level-triggered (LT) modes. * **`kqueue` (BSD, macOS):** Similar concept to `epoll`, providing efficient event notification for various event types (sockets, files, signals, timers).

While `poll`, `epoll`, and `kqueue` are often preferred for high-performance applications, understanding `select` and non-blocking fundamentals is essential, as the core concepts of handling `EWOULDBLOCK`/`EAGAIN`, partial I/O, and connection states remain the same.

9. Conclusion

Non-blocking sockets, combined with an I/O multiplexing mechanism like `select`, are fundamental tools for building responsive and scalable network applications in C. They allow a single thread to manage numerous connections concurrently without getting stuck waiting for any single operation.

The transition from blocking to non-blocking programming requires careful state management, meticulous error checking (especially for `EWOULDBLOCK`/`EAGAIN`, `EINPROGRESS`, `ECONNRESET`, `EPIPE`, and `EINTR`), handling partial reads/writes, and correctly using `select` (or its alternatives) to determine I/O readiness. While more complex initially, mastering non-blocking I/O unlocks the ability to create high-performance network services. Remember to always check return codes, consult `errno`, and handle all possible outcomes of non-blocking calls.