# Linux Server API — Overview

---

## 1. Creating and Binding a Socket

A server first creates a **listening socket** — a file descriptor representing a communication endpoint.

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
bind(sockfd, (struct sockaddr*)&addr, sizeof(addr));
listen(sockfd, BACKLOG);
```

**Meaning:**

- `socket()` — create a TCP or UDP endpoint
- `bind()` — attach it to an IP and port
- `listen()` — tell the kernel to queue incoming connections

By default, sockets are **blocking** — each operation (`accept`, `read`, `write`, etc.) waits until it can proceed.

---

## 2. Accepting Connections

The server waits for clients using:

```
int clientfd = accept(sockfd, (struct sockaddr*)&client, &len);
```

**Note:** `accept()` is a **blocking call** by default — it pauses until a client actually connects. If you want it to be **non-blocking**, you can set the socket with:

```
fcntl(sockfd, F_SETFL, O_NONBLOCK);
```

Then `accept()` will return immediately, with `-1` and `errno == EAGAIN` if no connection is waiting.

This **returns a new socket** for that specific client. Now the server can `read()` and `write()` on `clientfd`.

---

## 3. Handling Multiple Clients with `fork()`

To serve multiple clients:

- Without `fork()`, only one client is handled at a time.
- With `fork()`, each child process handles one connection independently.

```c
if (fork() == 0) {
    close(sockfd);
    handle_client(clientfd);
    exit(0);
}
```

Each child process has its own copy of the socket and runs in parallel. `read()` and `write()` on blocking sockets will also block until data is ready or sent.

---

## 4. Multiplexing with `select()`

Instead of using multiple processes, a server can ask the **kernel to monitor many sockets** using `select()`:

```c
fd_set readfds;            // a set (bitmap) of file descriptors to monitor
FD_ZERO(&readfds);         // clear the set (initialize all bits to 0)
FD_SET(sockfd, &readfds); // add our listening socket to the set
select(maxfd+1, &readfds, NULL, NULL, NULL); // wait until any socket is ready
```

**Idea:** the kernel tells us which sockets are ready for reading or writing. One process can handle many clients without threads or `fork()`.

**Blocking vs. Non-blocking:**

- `select()` itself **blocks** until one of the sockets becomes ready (unless you set a timeout).
- The sockets it monitors can still be **blocking**, but since you only act on ready ones, your code won't hang.

---

#### How `select()` Monitors Sockets

You still start by creating and binding a listening socket:

```c
int listenfd = socket(AF_INET, SOCK_STREAM, 0);
bind(listenfd, (struct sockaddr*)&addr, sizeof(addr));
listen(listenfd, BACKLOG);
```

Then you tell the kernel which sockets to monitor:

```c
fd_set master_set;          // stores all sockets we want to monitor
FD_ZERO(&master_set);       // initialize it (clear all file descriptors)
FD_SET(listenfd, &master_set); // add the listening socket to the set
int maxfd = listenfd;       // keep track of the highest-numbered file descriptor
```

In the main loop:

```c
fd_set readfds = master_set; // make a working copy (select modifies it)
select(maxfd + 1, &readfds, NULL, NULL, NULL); // block until activity occurs
```

```c
// loop through all possible sockets to see which ones are ready
for (int fd = 0; fd <= maxfd; fd++) {
    if (FD_ISSET(fd, &readfds)) { // true if fd is marked as ready by select()
        if (fd == listenfd) {
            // new connection available on the listening socket
            int clientfd = accept(listenfd, NULL, NULL); // may block if no connection (norm
            FD_SET(clientfd, &master_set); // add new client to the monitored set
            if (clientfd > maxfd) maxfd = clientfd; // update max fd number
        } else {
            // existing client sent data - handle it
            handle_client(fd);
        }
    }
}
```

When a client disconnects, remove it from the set:

```c
FD_CLR(fd, &master_set); // remove fd from master set so it's no longer watched
close(fd);               // free system resources
```

So the monitored pool gradually includes the listening socket and all connected clients. The **kernel** watches them all and wakes the process when any are ready.

---

### How `fd_set` Works Internally

- `fd_set` is a **bit mask structure** that tracks sockets by their file descriptor number. Each bit corresponds to one possible file descriptor — bit `n` is `1` if that socket is in the set.
- Functions like `FD_SET(fd, &set)` simply set that bit; `FD_CLR()` clears it; `FD_ISSET()` checks it.

**Limit:** the number of bits depends on `FD_SETSIZE`.

- On most Linux systems, `FD_SETSIZE` = **1024**, meaning you can monitor up to 1024 file descriptors.
- Each bit maps directly to a file descriptor number (0–1023 by default).

You can check it in code:

```c
#include <sys/select.h>
#include <stdio.h>
int main() {
    printf("FD_SETSIZE = %d\n", FD_SETSIZE);
    return 0;
}
```

If a program uses more than `FD_SETSIZE` sockets, `select()` cannot handle all of them — that's one reason `poll()` and `epoll()` exist.

---

## 5. `poll()` — A Better `select()`

`poll()` replaces file descriptor sets with an array, avoiding size limits:

```
struct pollfd fds[MAX];          // array of structures describing each socket
int nfds = 1;                    // number of sockets being watched
fds[0].fd = listenfd;            // first socket to monitor (the listener)
fds[0].events = POLLIN;          // we care about readable events (incoming data)

poll(fds, nfds, timeout);        // blocks until one or more sockets are ready

for (int i = 0; i < nfds; i++) {
    if (fds[i].revents & POLLIN) { // POLLIN means data is available
        if (fds[i].fd == listenfd) {
            // handle new incoming connection
        } else {
            // handle data from existing client
        }
    }
}
```

**Blocking vs. Non-blocking:** `poll()` behaves similarly to `select()` — it **blocks** until an event or timeout occurs. Sockets inside can still be blocking or non-blocking.

---

## 6. `epoll()` — Efficient Event Notification

For large servers (hundreds or thousands of connections), `epoll()` is most efficient:

```
int epfd = epoll_create1(0);                        // create epoll instance
struct epoll_event event;                           // describe event type
event.events = EPOLLIN;                             // interested in read events
event.data.fd = sockfd;                             // socket to monitor
epoll_ctl(epfd, EPOLL_CTL_ADD, sockfd, &event);     // register socket with epoll

struct epoll_event events[MAX_EVENTS];              // storage for triggered events
int n = epoll_wait(epfd, events, MAX_EVENTS, -1);   // block until some are ready

for (int i = 0; i < n; i++) {
    int fd = events[i].data.fd;                     // get the ready socket
```

```
    if (fd == sockfd) {
        // accept new connection
    } else {
        // handle data from connected client
    }
}
```

**Comments:**

- `epoll_create1()` makes a kernel-managed event table.
- `epoll_ctl()` registers, modifies, or removes sockets.
- `epoll_wait()` blocks until one or more events occur, returning only active ones.

**Blocking vs. Non-blocking:**

- `epoll_wait()` blocks until something happens (unless timeout = 0).
- Each socket can still be **blocking** or **non-blocking**, but most high-performance servers use **non-blocking sockets** with `epoll`.

`epoll()` scales with the number of **active** sockets, not total sockets.

---

**Summary — Blocking vs. Non-blocking**

| Function | Default Behavior | Non-blocking Possible? | Description |
|---|---|---|---|
| `accept()` | Blocking | Yes (`O_NONBLOCK`) | Waits for a new client connection |
| `read()` / `recv()` | Blocking | Yes | Waits for incoming data |
| `write()` / `send()` | Blocking | Yes | Waits for buffer space to send data |
| `select()` | Blocking | N/A (can set timeout) | Waits for any socket to become ready |
| `poll()` | Blocking | N/A (can set timeout) | Waits for events on multiple sockets |
| `epoll_wait()` | Blocking | N/A (can set timeout) | Waits for active events efficiently |

---

**Key takeaway:**

- By default, socket functions block until they can proceed.
- You can make sockets non-blocking to avoid waiting, and combine that with `select()`, `poll()`, or `epoll()` to handle many clients smoothly.