# Using epoll in Linux

`epoll` is a Linux system call designed for efficiently monitoring multiple file descriptors for events, such as when data is available to read or a socket is ready to write. It is particularly useful in network programming, where a server might need to handle many connections simultaneously. Compared to older mechanisms like `select` or `poll`, `epoll` scales better with a large number of file descriptors because it avoids repeatedly scanning all descriptors for readiness.

Here's a step-by-step guide to using `epoll` in Linux, followed by a simple example of a server that accepts connections and echoes back data.

---

## Steps to Use `epoll`

### 1. Create an Epoll Instance

To start using `epoll`, you need to create an epoll instance, which is represented by a file descriptor. Use the `epoll_create1` system call:

```
int epoll_fd = epoll_create1(0);
```

- **Parameter**: Pass `0` for basic usage (no special flags). Flags like `EPOLL_CLOEXEC` can be used for advanced scenarios.
- **Return Value**: Returns a file descriptor (`epoll_fd`) on success, or `-1` on error (check `errno` for details).

The older `epoll_create` function is similar but takes a size hint (now ignored), so `epoll_create1` is preferred.

### 2. Add File Descriptors to Monitor

Use `epoll_ctl` to register file descriptors (e.g., sockets) with the epoll instance and specify the events you want to monitor:

```
struct epoll_event ev;
ev.events = EPOLLIN;  // Monitor for readability
ev.data.fd = some_fd; // File descriptor to monitor
epoll_ctl(epoll_fd, EPOLL_CTL_ADD, some_fd, &ev);
```

- **Parameters**:
  - `epoll_fd`: The epoll instance file descriptor.
  - `EPOLL_CTL_ADD`: Operation to add a file descriptor.

- `some_fd`: The file descriptor to monitor (e.g., a socket).
  - `&ev`: Pointer to a `struct epoll_event` defining the events and optional user data.
- **Common Events**:
  - `EPOLLIN`: Data available to read.
  - `EPOLLOUT`: Ready to write.
  - `EPOLLERR`: Error occurred.
  - `EPOLLHUP`: Hang-up (e.g., connection closed).
- **User Data**: The `data` field in `struct epoll_event` can store a file descriptor (as shown) or other data (e.g., a pointer) to identify the source when events occur.

### 3. Wait for Events

Use `epoll_wait` to block and wait for events on the monitored file descriptors:

```
struct epoll_event events[MAX_EVENTS];
int nfds = epoll_wait(epoll_fd, events, MAX_EVENTS, -1);
```

- **Parameters**:
  - `epoll_fd`: The epoll instance.
  - `events`: Array to store triggered events.
  - `MAX_EVENTS`: Maximum number of events to return (size of the array).
  - `-1`: Timeout in milliseconds (`-1` means wait indefinitely; `0` returns immediately).
- **Return Value**: Number of file descriptors with events (`nfds`), or `-1` on error.

### 4. Handle Events

Loop through the events returned by `epoll_wait` and process them:

```
for (int i = 0; i < nfds; i++) {
    if (events[i].events & EPOLLIN) {
        // File descriptor events[i].data.fd is readable
    }
}
```

- Check the `events` field using bitwise operations (e.g., `events[i].events & EPOLLIN`) to determine the event type.
- Use `events[i].data.fd` to identify which file descriptor triggered the event.

#### 5. Manage File Descriptors (Optional)

- **Remove**: Use `epoll_ctl` with `EPOLL_CTL_DEL` to stop monitoring a file descriptor:

  ```
  epoll_ctl(epoll_fd, EPOLL_CTL_DEL, some_fd, NULL);
  ```

- **Modify**: Adjust events with `EPOLL_CTL_MOD`:

  ```
  ev.events = EPOLLOUT; // Change to monitor writability
  epoll_ctl(epoll_fd, EPOLL_CTL_MOD, some_fd, &ev);
  ```

---

## Key Concepts

### Level-Triggered vs. Edge-Triggered

- **Level-Triggered (Default)**: `epoll` notifies repeatedly as long as the condition persists (e.g., data remains unread). Simpler for most cases.
- **Edge-Triggered (`EPOLLET`)**: Notifies only once when the state changes (e.g., new data arrives). Requires reading/writing all data until `EAGAIN` to avoid missing events; more efficient but trickier.
- Set `EPOLLET` in `ev.events` (e.g., `EPOLLIN | EPOLLET`) if using edge-triggered mode.

### Non-Blocking I/O

`epoll` is often paired with non-blocking file descriptors to prevent blocking on I/O operations. Set a socket to non-blocking mode with:

```
fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) | O_NONBLOCK);
```

---

## Example: Simple Echo Server

Below is a basic example of a server that uses `epoll` to accept connections and echo data back to clients. It uses level-triggered mode for simplicity.

```
#include <sys/epoll.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <fcntl.h>
#include <unistd.h>
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define MAX_EVENTS 10
#define PORT 8080

int main() {
    // Create listening socket
    int listen_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (listen_fd == -1) { perror("socket"); exit(1); }

    struct sockaddr_in addr = { .sin_family = AF_INET, .sin_addr.s_addr = INADDR_ANY, .sin_port = htons(PORT)
    if (bind(listen_fd, (struct sockaddr*)&addr, sizeof(addr)) == -1) { perror("bind"); exit(1); }
    if (listen(listen_fd, 5) == -1) { perror("listen"); exit(1); }

    // Set listening socket to non-blocking
    fcntl(listen_fd, F_SETFL, fcntl(listen_fd, F_GETFL) | O_NONBLOCK);

    // Create epoll instance
    int epoll_fd = epoll_create1(0);
    if (epoll_fd == -1) { perror("epoll_create1"); exit(1); }

    // Add listening socket to epoll
    struct epoll_event ev, events[MAX_EVENTS];
    ev.events = EPOLLIN; // Level-triggered
    ev.data.fd = listen_fd;
    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, listen_fd, &ev) == -1) { perror("epoll_ctl"); exit(1); }

    // Event loop
    while (1) {
        int nfds = epoll_wait(epoll_fd, events, MAX_EVENTS, -1);
        if (nfds == -1) { perror("epoll_wait"); exit(1); }

        for (int i = 0; i < nfds; i++) {
            int fd = events[i].data.fd;

            if (fd == listen_fd) {
                // Accept new connection
                int client_fd = accept(listen_fd, NULL, NULL);
```

```c
                if (client_fd == -1) { perror("accept"); continue; }

                // Set client socket to non-blocking
                fcntl(client_fd, F_SETFL, fcntl(client_fd, F_GETFL) | O_NONBLOCK);

                // Add client socket to epoll
                ev.events = EPOLLIN;
                ev.data.fd = client_fd;
                if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, client_fd, &ev) == -1) {
                    perror("epoll_ctl: client_fd");
                    close(client_fd);
                }
            } else {
                // Handle client data
                char buf[1024];
                ssize_t n = read(fd, buf, sizeof(buf));
                if (n <= 0) {
                    // Error or EOF: close and remove from epoll
                    if (n == -1 && (errno == EAGAIN || errno == EWOULDBLOCK)) continue;
                    close(fd);
                    epoll_ctl(epoll_fd, EPOLL_CTL_DEL, fd, NULL);
                } else {
                    // Echo data back
                    write(fd, buf, n); // Note: In real apps, handle partial writes
                }
            }
        }
    }

    close(epoll_fd);
    close(listen_fd);
    return 0;
}
```

### Notes on the Example

- **Listening Socket**: Monitors `EPOLLIN` to detect new connections.
- **Client Sockets**: Also monitor `EPOLLIN` to detect incoming data.
- **Simplification**: Assumes `write` completes fully. In production, buffer data and use `EPOLLOUT` for partial writes.

- **Error Handling**: Closes sockets on errors or EOF and removes them from epoll.

---

## Summary

`epoll` provides an efficient way to handle multiple file descriptors in Linux: 1. Create an instance with `epoll_create1`. 2. Register file descriptors and events with `epoll_ctl`. 3. Wait for events with `epoll_wait`. 4. Process events in a loop, adjusting monitored events or removing descriptors as needed.

For simple applications, level-triggered mode is recommended. For high-performance needs, consider edge-triggered mode with careful handling of all available data. Always pair `epoll` with non-blocking I/O for best results.