# 使用 epoll 在 Linux

epoll 是一个设计用于高效监控多个文件描述符事件的 Linux 系统调用，例如数据可读或套接字可写。它在网络编程中特别有用，服务器可能需要同时处理许多连接。与 select 或 poll 等较旧机制相比，epoll 可以更好地处理大量文件描述符，因为它避免了重复扫描所有描述符的准备状态。

以下是在 Linux 中使用 epoll 的分步指南，后面是一个接受连接并回显数据的简单服务器示例。

---

## 使用 epoll 的步骤

### 1. 创建一个 Epoll 实例

要开始使用 epoll，需要创建一个 epoll 实例，它由一个文件描述符表示。使用 epoll_create1 系统调用：

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

- **参数**：传递 0 以进行基本使用（无特殊标志）。可以使用 EPOLL_CLOEXEC 等标志进行高级场景。
- **返回值**：成功时返回文件描述符（epoll_fd），错误时返回 -1（检查 errno 以获取详细信息）。

较旧的 epoll_create 函数类似，但接受一个大小提示（现在被忽略），因此优先使用 epoll_create1。

### 2. 添加要监控的文件描述符

使用 epoll_ctl 将文件描述符（例如，套接字）注册到 epoll 实例，并指定要监控的事件：

```
struct epoll_event ev;
ev.events = EPOLLIN;  // 监控可读性
ev.data.fd = some_fd; // 要监控的文件描述符
epoll_ctl(epoll_fd, EPOLL_CTL_ADD, some_fd, &ev);
```

- **参数**：
  - epoll_fd：epoll 实例文件描述符。
  - EPOLL_CTL_ADD：添加文件描述符的操作。
  - some_fd：要监控的文件描述符（例如，套接字）。
  - &ev：指向定义事件和可选用户数据的 struct epoll_event。
- **常见事件**：
  - EPOLLIN：数据可读。
  - EPOLLOUT：可写。

  - EPOLLERR：发生错误。
  - EPOLLHUP：挂断（例如，连接关闭）。
- **用户数据**：struct epoll_event 中的 data 字段可以存储文件描述符（如上所示）或其他数据（例如，指针），以在事件发生时识别源。

### 3. 等待事件

使用 epoll_wait 阻塞并等待监控文件描述符上的事件：

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

- **参数**：
  - epoll_fd：epoll 实例。
  - events：存储触发事件的数组。
  - MAX_EVENTS：要返回的最大事件数（数组大小）。
  - -1：超时时间（毫秒）（-1 表示无限期等待；0 立即返回）。
- **返回值**：具有事件的文件描述符数量（nfds），或错误时返回 -1。

### 4. 处理事件

遍历 epoll_wait 返回的事件并处理它们：

```
for (int i = 0; i < nfds; i++) {
    if (events[i].events & EPOLLIN) {
        // 文件描述符 events[i].data.fd 可读
    }
}
```

- 使用位运算（例如，events[i].events & EPOLLIN）检查 events 字段，以确定事件类型。
- 使用 events[i].data.fd 识别触发事件的文件描述符。

### 5. 管理文件描述符（可选）

- **删除**：使用 epoll_ctl 与 EPOLL_CTL_DEL 停止监控文件描述符：

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

- **修改**：使用 EPOLL_CTL_MOD 调整事件：

  ```
  ev.events = EPOLLOUT; // 更改为监控可写性
  epoll_ctl(epoll_fd, EPOLL_CTL_MOD, some_fd, &ev);
  ```

## 关键概念

### 水平触发 vs. 边缘触发

- **水平触发（默认）**：`epoll` 在条件持续存在时（例如，数据未读）重复通知。对于大多数情况更简单。
- **边缘触发（`EPOLLET`）**：在状态更改时（例如，新数据到达）仅通知一次。需要读取/写入所有数据，直到 `EAGAIN` 以避免丢失事件；更高效但更复杂。
- 如果使用边缘触发模式，请在 `ev.events` 中设置 `EPOLLET`（例如，`EPOLLIN | EPOLLET`）。

### 非阻塞 I/O

`epoll` 通常与非阻塞文件描述符配对，以防止在 I/O 操作上阻塞。使用以下方法将套接字设置为非阻塞模式：

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

---

## 示例：简单回显服务器

以下是一个使用 `epoll` 接受连接并将数据回显给客户端的基本服务器示例。它使用水平触发模式以简化。

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

#define MAX_EVENTS 10
#define PORT 8080

int main() {
    // 创建监听套接字
    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); }
```

```
// 将监听套接字设置为非阻塞
fcntl(listen_fd, F_SETFL, fcntl(listen_fd, F_GETFL) | O_NONBLOCK);

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

// 将监听套接字添加到 epoll
struct epoll_event ev, events[MAX_EVENTS];
ev.events = EPOLLIN; // 水平触发
ev.data.fd = listen_fd;
if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, listen_fd, &ev) == -1) { perror("epoll_ctl"); exit(1); }

// 事件循环
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) {
            // 接受新连接
            int client_fd = accept(listen_fd, NULL, NULL);
            if (client_fd == -1) { perror("accept"); continue; }

            // 将客户端套接字设置为非阻塞
            fcntl(client_fd, F_SETFL, fcntl(client_fd, F_GETFL) | O_NONBLOCK);

            // 将客户端套接字添加到 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 {
            // 处理客户端数据
            char buf[1024];
```

```
                ssize_t n = read(fd, buf, sizeof(buf));
                if (n <= 0) {
                    // 错误或 EOF：关闭并从 epoll 中删除
                    if (n == -1 && (errno == EAGAIN || errno == EWOULDBLOCK)) continue;
                    close(fd);
                    epoll_ctl(epoll_fd, EPOLL_CTL_DEL, fd, NULL);
                } else {
                    // 回显数据
                    write(fd, buf, n); // 注意：在实际应用中处理部分写入
                }
            }
        }
    }

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

**示例说明**

- **监听套接字**：监控 EPOLLIN 以检测新连接。
- **客户端套接字**：也监控 EPOLLIN 以检测传入数据。
- **简化**：假设 write 完全完成。在生产中，缓冲数据并使用 EPOLLOUT 进行部分写入。
- **错误处理**：在错误或 EOF 时关闭套接字并从 epoll 中删除。

---

## 总结

epoll 提供了一种高效处理 Linux 中多个文件描述符的方法：1. 使用 epoll_create1 创建实例。2. 使用 epoll_ctl 注册文件描述符和事件。3. 使用 epoll_wait 等待事件。4. 在循环中处理事件，根据需要调整监控事件或删除描述符。

对于简单应用程序，建议使用水平触发模式。对于高性能需求，考虑使用边缘触发模式，并仔细处理所有可用数据。始终将 epoll 与非阻塞 I/O 配对以获得最佳结果。