Verwendung von epoll in Linux

`epoll` ist ein Linux-Systemaufruf, der für das effiziente Überwachen mehrerer Dateideskriptoren auf Ereignisse entwickelt wurde, wie z.B. wenn Daten zum Lesen verfügbar sind oder ein Socket zum Schreiben bereit ist. Es ist besonders nützlich in der Netzwerkprogrammierung, wo ein Server möglicherweise viele Verbindungen gleichzeitig verwalten muss. Im Vergleich zu älteren Mechanismen wie `select` oder `poll` skaliert `epoll` besser mit einer großen Anzahl von Dateideskriptoren, da es das wiederholte Scannen aller Deskriptoren auf Bereitschaft vermeidet.

Hier ist eine Schritt-für-Schritt-Anleitung zur Verwendung von `epoll` in Linux, gefolgt von einem einfachen Beispiel eines Servers, der Verbindungen akzeptiert und Daten zurücksendet.

Schritte zur Verwendung von `epoll`

1. Erstellen einer Epoll-Instanz

Um `epoll` zu verwenden, müssen Sie eine Epoll-Instanz erstellen, die durch einen Dateideskriptor dargestellt wird. Verwenden Sie den Systemaufruf `epoll_create1`:

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

- **Parameter**: Geben Sie `0` für die grundlegende Verwendung (keine speziellen Flags) ein. Flags wie `EPOLL_CLOEXEC` können für fortgeschrittene Szenarien verwendet werden.
- **Rückgabewert**: Gibt einen Dateideskriptor (`epoll_fd`) bei Erfolg oder `-1` bei einem Fehler (überprüfen Sie `errno` für Details) zurück.

Die ältere Funktion `epoll_create` ist ähnlich, nimmt jedoch eine Größenangabe (jetzt ignoriert), sodass `epoll_create1` bevorzugt wird.

2. Hinzufügen von Dateideskriptoren zur Überwachung

Verwenden Sie `epoll_ctl`, um Dateideskriptoren (z.B. Sockets) mit der Epoll-Instanz zu registrieren und die Ereignisse anzugeben, die Sie überwachen möchten:

```
struct epoll_event ev;
ev.events = EPOLLIN;  // Überwachen auf Lesbarkeit
ev.data.fd = some_fd; // Zu überwachender Dateideskriptor
epoll_ctl(epoll_fd, EPOLL_CTL_ADD, some_fd, &ev);
```

- **Parameter**:

  - `epoll_fd`: Der Dateideskriptor der Epoll-Instanz.
  - `EPOLL_CTL_ADD`: Operation zum Hinzufügen eines Dateideskriptors.
  - `some_fd`: Der zu überwachende Dateideskriptor (z.B. ein Socket).
  - `&ev`: Zeiger auf eine `struct epoll_event`, die die Ereignisse und optionale Benutzerdaten definiert.
- **Gängige Ereignisse**:
  - `EPOLLIN`: Daten sind zum Lesen verfügbar.
  - `EPOLLOUT`: Bereit zum Schreiben.
  - `EPOLLERR`: Fehler aufgetreten.
  - `EPOLLHUP`: Auflegen (z.B. Verbindung geschlossen).
- **Benutzerdaten**: Das Feld `data` in `struct epoll_event` kann einen Dateideskriptor (wie gezeigt) oder andere Daten (z.B. einen Zeiger) speichern, um die Quelle zu identifizieren, wenn Ereignisse auftreten.

3. Warten auf Ereignisse

Verwenden Sie `epoll_wait`, um zu blockieren und auf Ereignisse auf den überwachten Dateideskriptoren zu warten:

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

- **Parameter**:
  - `epoll_fd`: Die Epoll-Instanz.
  - `events`: Array zum Speichern ausgelöster Ereignisse.
  - `MAX_EVENTS`: Maximale Anzahl von Ereignissen, die zurückgegeben werden sollen (Größe des Arrays).
  - `-1`: Timeout in Millisekunden (`-1` bedeutet, warten Sie unendlich; `0` gibt sofort zurück).
- **Rückgabewert**: Anzahl der Dateideskriptoren mit Ereignissen (`nfds`) oder `-1` bei einem Fehler.

4. Ereignisse verarbeiten

Durchlaufen Sie die von `epoll_wait` zurückgegebenen Ereignisse und verarbeiten Sie sie:

```
for (int i = 0; i < nfds; i++) {
    if (events[i].events & EPOLLIN) {
        // Dateideskriptor events[i].data.fd ist lesbar
    }
}
```

- Überprüfen Sie das Feld `events` mit bitweisen Operationen (z.B. `events[i].events & EPOLLIN`), um den Ereignistyp zu bestimmen.

- Verwenden Sie `events[i].data.fd`, um zu identifizieren, welcher Dateideskriptor das Ereignis ausgelöst hat.

5. Verwalten von Dateideskriptoren (optional)

- **Entfernen**: Verwenden Sie `epoll_ctl` mit `EPOLL_CTL_DEL`, um das Überwachen eines Dateideskriptors zu stoppen:

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

- **Ändern**: Passen Sie Ereignisse mit `EPOLL_CTL_MOD` an:

```
ev.events = EPOLLOUT; // Ändern, um Schreibbereitschaft zu überwachen
epoll_ctl(epoll_fd, EPOLL_CTL_MOD, some_fd, &ev);
```

Wichtige Konzepte

Level-Triggered vs. Edge-Triggered

- **Level-Triggered (Standard)**: `epoll` benachrichtigt wiederholt, solange die Bedingung besteht (z.B. Daten bleiben ungelesen). Einfacher für die meisten Fälle.
- **Edge-Triggered (`EPOLLET`)**: Benachrichtigt nur einmal, wenn sich der Zustand ändert (z.B. neue Daten eintreffen). Erfordert das Lesen/Schreiben aller Daten bis `EAGAIN`, um das Verpassen von Ereignissen zu vermeiden; effizienter, aber kniffliger.
- Setzen Sie `EPOLLET` in `ev.events` (z.B. `EPOLLIN | EPOLLET`), wenn Sie den Edge-Triggered-Modus verwenden.

Non-Blocking I/O

`epoll` wird oft mit nicht blockierenden Dateideskriptoren kombiniert, um das Blockieren bei I/O-Vorgängen zu verhindern. Setzen Sie einen Socket in den nicht blockierenden Modus mit:

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

Beispiel: Einfacher Echo-Server

Hier ist ein grundlegendes Beispiel eines Servers, der `epoll` verwendet, um Verbindungen zu akzeptieren und Daten an die Clients zurückzusenden. Es verwendet den Level-Triggered-Modus aus Gründen der Einfachheit.

```
#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() {
    // Erstellen Sie einen 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); }

    // Setzen Sie den Listening-Socket auf nicht blockierend
    fcntl(listen_fd, F_SETFL, fcntl(listen_fd, F_GETFL) | O_NONBLOCK);

    // Erstellen Sie eine Epoll-Instanz
    int epoll_fd = epoll_create1(0);
    if (epoll_fd == -1) { perror("epoll_create1"); exit(1); }

    // Fügen Sie den Listening-Socket zu Epoll hinzu
    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); }

    // Ereignis-Schleife
```

```
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) {
            // Akzeptieren Sie eine neue Verbindung
            int client_fd = accept(listen_fd, NULL, NULL);
            if (client_fd == -1) { perror("accept"); continue; }

            // Setzen Sie den Client-Socket auf nicht blockierend
            fcntl(client_fd, F_SETFL, fcntl(client_fd, F_GETFL) | O_NONBLOCK);

            // Fügen Sie den Client-Socket zu Epoll hinzu
            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 {
            // Verarbeiten Sie Client-Daten
            char buf[1024];
            ssize_t n = read(fd, buf, sizeof(buf));
            if (n <= 0) {
                // Fehler oder EOF: schließen und aus Epoll entfernen
                if (n == -1 && (errno == EAGAIN || errno == EWOULDBLOCK)) continue;
                close(fd);
                epoll_ctl(epoll_fd, EPOLL_CTL_DEL, fd, NULL);
            } else {
                // Daten zurücksenden
                write(fd, buf, n); // Hinweis: In echten Anwendungen Teilschreibvorgänge behandeln
            }
        }
    }
}

close(epoll_fd);
```

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

Hinweise zum Beispiel

- **Listening-Socket**: Überwacht `EPOLLIN`, um neue Verbindungen zu erkennen.
- **Client-Sockets**: Überwachen ebenfalls `EPOLLIN`, um eingehende Daten zu erkennen.
- **Vereinfachung**: Geht davon aus, dass `write` vollständig abgeschlossen wird. In der Produktion sollten Daten gepuffert und `EPOLLOUT` für Teilschreibvorgänge verwendet werden.
- **Fehlerbehandlung**: Schließt Sockets bei Fehlern oder EOF und entfernt sie aus Epoll.

Zusammenfassung

`epoll` bietet eine effiziente Möglichkeit, mehrere Dateideskriptoren in Linux zu verwalten: 1. Erstellen Sie eine Instanz mit `epoll_create1`. 2. Registrieren Sie Dateideskriptoren und Ereignisse mit `epoll_ctl`. 3. Warten Sie auf Ereignisse mit `epoll_wait`. 4. Verarbeiten Sie Ereignisse in einer Schleife und passen Sie die überwachten Ereignisse oder entfernen Sie Deskriptoren nach Bedarf an.

Für einfache Anwendungen wird der Level-Triggered-Modus empfohlen. Für hohe Leistungsanforderungen sollten Sie den Edge-Triggered-Modus mit sorgfältiger Handhabung aller verfügbaren Daten in Betracht ziehen. Verwenden Sie `epoll` immer in Kombination mit nicht blockierendem I/O für die besten Ergebnisse.