# epoll を Linux で使用する

epoll は、Linux システムコールで、複数のファイルディスクリプタをイベント（例えば、読み取り可能なデータが利用可能か、ソケットが書き込み可能か）を効率的に監視するために設計されています。特にネットワークプログラミングで、サーバーが多くの接続を同時に処理する必要がある場合に有用です。古いメカニズムである select や poll に比べて、epoll は、ファイルディスクリプタの数が多い場合にスケールが良いです。これは、すべてのディスクリプタを準備状態を確認するために繰り返しスキャンする必要がないためです。

以下は、Linux で epoll を使用するためのステップバイステップのガイドと、接続を受け入れてデータをエコーバックする簡単なサーバーの例です。

---

## epoll の使用方法

### 1. Epoll インスタンスの作成

epoll を使用するには、ファイルディスクリプタで表される epoll インスタンスを作成する必要があります。epoll_create1 システムコールを使用します。

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

- **パラメータ**: 基本的な使用には 0 を渡します（特別なフラグはありません）。高度なシナリオでは EPOLL_CLOEXEC のようなフラグを使用できます。
- **戻り値**: 成功するとファイルディスクリプタ (epoll_fd) を返し、エラーの場合は -1 を返します（詳細については errno を確認）。

古い epoll_create 関数は似ていますが、サイズのヒントを取るため、epoll_create1 が推奨されます。

### 2. 監視するファイルディスクリプタの追加

epoll_ctl を使用して、ファイルディスクリプタ（例えば、ソケット）を epoll インスタンスに登録し、監視したいイベントを指定します。

```
struct epoll_event ev;
ev.events = EPOLLIN;  // 読み取り可能を監視
ev.data.fd = some_fd; // 監視するファイルディスクリプタ
epoll_ctl(epoll_fd, EPOLL_CTL_ADD, some_fd, &ev);
```

- **パラメータ**:
  - epoll_fd: epoll インスタンスのファイルディスクリプタ。

  - EPOLL_CTL_ADD: ファイルディスクリプタを追加する操作。
  - some_fd: 監視するファイルディスクリプタ（例えば、ソケット）。
  - &ev: イベントとオプションのユーザーデータを定義する struct epoll_event へのポインタ。
- **一般的なイベント**:
  - EPOLLIN: 読み取り可能なデータ。
  - EPOLLOUT: 書き込み可能。
  - EPOLLERR: エラーが発生。
  - EPOLLHUP: ハングアップ（例えば、接続が閉じる）。
- **ユーザーデータ**: struct epoll_event の data フィールドには、ファイルディスクリプタ（上記のように）または他のデータ（例えば、ポインタ）を格納して、イベントが発生したときにソースを識別できます。

### 3. イベントの待機

epoll_wait を使用して、監視中のファイルディスクリプタでイベントをブロックして待機します。

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

- **パラメータ**:
  - epoll_fd: epoll インスタンス。
  - events: トリガーされたイベントを格納する配列。
  - MAX_EVENTS: 返すイベントの最大数（配列のサイズ）。
  - -1: ミリ秒単位のタイムアウト（-1 は無限に待機、0 は即座に返す）。
- **戻り値**: イベントがあるファイルディスクリプタの数 (nfds) またはエラーの場合は -1。

### 4. イベントの処理

epoll_wait によって返されたイベントをループして処理します。

```
for (int i = 0; i < nfds; i++) {
    if (events[i].events & EPOLLIN) {
        // ファイルディスクリプタ events[i].data.fd は読み取り可能
    }
}
```

- ビット演算（例えば、events[i].events & EPOLLIN）を使用して events フィールドを確認し、イベントの種類を決定します。
- events[i].data.fd を使用して、どのファイルディスクリプタがイベントをトリガーしたかを特定します。

### 5. ファイルディスクリプタの管理（オプション）

- **削除**: `epoll_ctl` を `EPOLL_CTL_DEL` と一緒に使用して、ファイルディスクリプタの監視を停止します。

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

- **修正**: `EPOLL_CTL_MOD` を使用してイベントを調整します。

```
ev.events = EPOLLOUT; // 書き込み可能を監視に変更
epoll_ctl(epoll_fd, EPOLL_CTL_MOD, some_fd, &ev);
```

---

## 主要な概念

### レベルトリガー vs. エッジトリガー

- **レベルトリガー（デフォルト）**: `epoll` は、条件が継続する限り、繰り返し通知します（例えば、データが未読のまま）。大多数の場合に簡単です。
- **エッジトリガー（`EPOLLET`）**: 状態が変化したときにのみ通知します（例えば、新しいデータが到着）。すべてのデータを読み取り/書き込みするまで `EAGAIN` になるまで、イベントを欠かさないようにする必要があります。効率的ですが、やや難しいです。
- エッジトリガーモードを使用する場合は、`ev.events` に `EPOLLET` を設定します（例えば、`EPOLLIN | EPOLLET`）。

### 非ブロッキング I/O

`epoll` は、I/O 操作でブロックしないようにするために、非ブロッキングファイルディスクリプタと組み合わせることが一般的です。ソケットを非ブロッキングモードに設定するには、以下のようにします。

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

---

## 例: 簡単なエコーサーバー

以下は、`epoll` を使用して接続を受け入れ、クライアントにデータをエコーバックするサーバーの基本的な例です。簡単のためにレベルトリガーモードを使用しています。

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

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

#define MAX_EVENTS 10
#define PORT 8080

int main() {
    // リスニングソケットの作成
    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); }

    // リスニングソケットを非ブロッキングに設定
    fcntl(listen_fd, F_SETFL, fcntl(listen_fd, F_GETFL) | O_NONBLOCK);

    // epoll インスタンスの作成
    int epoll_fd = epoll_create1(0);
    if (epoll_fd == -1) { perror("epoll_create1"); exit(1); }

    // リスニングソケットを epoll に追加
    struct epoll_event ev, events[MAX_EVENTS];
    ev.events = EPOLLIN; // レベルトリガー
    ev.data.fd = listen_fd;
    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, listen_fd, &ev) == -1) { perror("epoll_ctl"); exit(1); }

    // イベントループ
    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) {
```

```
                // 新しい接続を受け入れる
                int client_fd = accept(listen_fd, NULL, NULL);
                if (client_fd == -1) { perror("accept"); continue; }

                // クライアントソケットを非ブロッキングに設定
                fcntl(client_fd, F_SETFL, fcntl(client_fd, F_GETFL) | O_NONBLOCK);

                // クライアントソケットを epoll に追加
                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 {
                // クライアントデータの処理
                char buf[1024];
                ssize_t n = read(fd, buf, sizeof(buf));
                if (n <= 0) {
                    // エラーまたは EOF: 閉じて epoll から削除
                    if (n == -1 && (errno == EAGAIN || errno == EWOULDBLOCK)) continue;
                    close(fd);
                    epoll_ctl(epoll_fd, EPOLL_CTL_DEL, fd, NULL);
                } else {
                    // データをエコーバック
                    write(fd, buf, n); // 注意: 実際のアプリケーションでは、部分的な書き込みを処理
                }
            }
        }
    }

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

**例の注意点**

- **リスニングソケット**: 新しい接続を検出するために `EPOLLIN` を監視します。
- **クライアントソケット**: 受信データを検出するために `EPOLLIN` を監視します。

- **簡略化**: `write` が完全に完了することを前提としています。本番環境では、バッファリングされたデータを使用し、`EPOLLOUT` を部分的な書き込みに使用します。
- **エラーハンドリング**: エラーまたは EOF の場合、ソケットを閉じて epoll から削除します。

---

## まとめ

`epoll` は、Linux で複数のファイルディスクリプタを効率的に処理する方法を提供します: 1. `epoll_create1` を使用してインスタンスを作成します。2. `epoll_ctl` を使用してファイルディスクリプタとイベントを登録します。3. `epoll_wait` を使用してイベントを待機します。4. イベントをループして処理し、必要に応じて監視中のイベントを調整またはディスクリプタを削除します。

簡単なアプリケーションには、レベルトリガーモードが推奨されます。高性能が必要な場合は、すべての利用可能なデータを慎重に処理するエッジトリガーモードを検討してください。常に `epoll` を非ブロッキング I/O と組み合わせて最適な結果を得るようにします。