Usando epoll en Linux

`epoll` es una llamada del sistema de Linux diseñada para monitorear de manera eficiente múltiples descriptores de archivos para eventos, como cuando los datos están disponibles para leer o un socket está listo para escribir. Es particularmente útil en la programación de redes, donde un servidor podría necesitar manejar muchas conexiones simultáneamente. En comparación con mecanismos más antiguos como `select` o `poll`, `epoll` escala mejor con un gran número de descriptores de archivos porque evita escanear repetidamente todos los descriptores para su disponibilidad.

Aquí tienes una guía paso a paso para usar `epoll` en Linux, seguida de un ejemplo simple de un servidor que acepta conexiones y devuelve los datos.

Pasos para usar `epoll`

1. Crear una instancia de epoll

Para comenzar a usar `epoll`, necesitas crear una instancia de epoll, que se representa por un descriptor de archivo. Usa la llamada del sistema `epoll_create1`:

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

- **Parámetro**: Pasa `0` para uso básico (sin banderas especiales). Banderas como `EPOLL_CLOEXEC` se pueden usar para escenarios avanzados.
- **Valor de retorno**: Devuelve un descriptor de archivo (`epoll_fd`) en caso de éxito, o `-1` en caso de error (verifica `errno` para detalles).

La función más antigua `epoll_create` es similar pero toma una pista de tamaño (ahora ignorada), por lo que `epoll_create1` es preferible.

2. Agregar descriptores de archivos para monitorear

Usa `epoll_ctl` para registrar descriptores de archivos (por ejemplo, sockets) con la instancia de epoll y especificar los eventos que deseas monitorear:

```
struct epoll_event ev;
ev.events = EPOLLIN;  // Monitorear para lectura
ev.data.fd = some_fd; // Descriptor de archivo a monitorear
epoll_ctl(epoll_fd, EPOLL_CTL_ADD, some_fd, &ev);
```

- **Parámetros**:

  - `epoll_fd`: El descriptor de archivo de la instancia de epoll.
  - `EPOLL_CTL_ADD`: Operación para agregar un descriptor de archivo.
  - `some_fd`: El descriptor de archivo a monitorear (por ejemplo, un socket).
  - `&ev`: Puntero a una `struct epoll_event` que define los eventos y los datos opcionales del usuario.
- **Eventos comunes**:
  - `EPOLLIN`: Datos disponibles para leer.
  - `EPOLLOUT`: Listo para escribir.
  - `EPOLLERR`: Ocurrió un error.
  - `EPOLLHUP`: Cierre (por ejemplo, conexión cerrada).
- **Datos del usuario**: El campo `data` en `struct epoll_event` puede almacenar un descriptor de archivo (como se muestra) u otros datos (por ejemplo, un puntero) para identificar la fuente cuando ocurran eventos.

3. Esperar eventos

Usa `epoll_wait` para bloquear y esperar eventos en los descriptores de archivos monitoreados:

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

- **Parámetros**:
  - `epoll_fd`: La instancia de epoll.
  - `events`: Arreglo para almacenar eventos desencadenados.
  - `MAX_EVENTS`: Número máximo de eventos para devolver (tamaño del arreglo).
  - `-1`: Tiempo de espera en milisegundos (`-1` significa esperar indefinidamente; `0` devuelve inmediatamente).
- **Valor de retorno**: Número de descriptores de archivos con eventos (`nfds`), o `-1` en caso de error.

4. Manejar eventos

Recorre los eventos devueltos por `epoll_wait` y procésalos:

```
for (int i = 0; i < nfds; i++) {
    if (events[i].events & EPOLLIN) {
        // El descriptor de archivo events[i].data.fd es legible
    }
}
```

- Verifica el campo `events` usando operaciones bit a bit (por ejemplo, `events[i].events & EPOLLIN`) para determinar el tipo de evento.
- Usa `events[i].data.fd` para identificar qué descriptor de archivo desencadenó el evento.

5. Administrar descriptores de archivos (opcional)

- **Eliminar**: Usa `epoll_ctl` con `EPOLL_CTL_DEL` para dejar de monitorear un descriptor de archivo:

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

- **Modificar**: Ajusta eventos con `EPOLL_CTL_MOD`:

```
ev.events = EPOLLOUT; // Cambiar para monitorear la capacidad de escritura
epoll_ctl(epoll_fd, EPOLL_CTL_MOD, some_fd, &ev);
```

Conceptos clave

Nivel disparado vs. Borde disparado

- **Nivel disparado (predeterminado)**: `epoll` notifica repetidamente mientras persista la condición (por ejemplo, los datos permanecen sin leer). Más sencillo para la mayoría de los casos.
- **Borde disparado (`EPOLLET`)**: Notifica solo una vez cuando cambia el estado (por ejemplo, llegan nuevos datos). Requiere leer/escribir todos los datos hasta `EAGAIN` para evitar perder eventos; más eficiente pero más complicado.
- Establece `EPOLLET` en `ev.events` (por ejemplo, `EPOLLIN | EPOLLET`) si usas el modo disparado por borde.

E/S no bloqueante

`epoll` a menudo se empareja con descriptores de archivos no bloqueantes para evitar bloquear operaciones de E/S. Establece un socket en modo no bloqueante con:

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

Ejemplo: Servidor de eco simple

A continuación, se muestra un ejemplo básico de un servidor que usa `epoll` para aceptar conexiones y devolver datos a los clientes. Usa el modo disparado por nivel para simplicidad.

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

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

#define MAX_EVENTS 10
#define PORT 8080

int main() {
    // Crear socket de escucha
    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); }

    // Establecer socket de escucha en modo no bloqueante
    fcntl(listen_fd, F_SETFL, fcntl(listen_fd, F_GETFL) | O_NONBLOCK);

    // Crear instancia de epoll
    int epoll_fd = epoll_create1(0);
    if (epoll_fd == -1) { perror("epoll_create1"); exit(1); }

    // Agregar socket de escucha a epoll
    struct epoll_event ev, events[MAX_EVENTS];
    ev.events = EPOLLIN; // Disparado por nivel
    ev.data.fd = listen_fd;
    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, listen_fd, &ev) == -1) { perror("epoll_ctl"); exit(1); }

    // Bucle de eventos
    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) {
                // Aceptar nueva conexión
```

```
                int client_fd = accept(listen_fd, NULL, NULL);
                if (client_fd == -1) { perror("accept"); continue; }

                // Establecer socket del cliente en modo no bloqueante
                fcntl(client_fd, F_SETFL, fcntl(client_fd, F_GETFL) | O_NONBLOCK);

                // Agregar socket del cliente a 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 {
                // Manejar datos del cliente
                char buf[1024];
                ssize_t n = read(fd, buf, sizeof(buf));
                if (n <= 0) {
                    // Error o EOF: cerrar y eliminar de epoll
                    if (n == -1 && (errno == EAGAIN || errno == EWOULDBLOCK)) continue;
                    close(fd);
                    epoll_ctl(epoll_fd, EPOLL_CTL_DEL, fd, NULL);
                } else {
                    // Devolver datos
                    write(fd, buf, n); // Nota: En aplicaciones reales, maneja escrituras parciales
                }
            }
        }
    }

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

Notas sobre el ejemplo

- **Socket de escucha**: Monitorea `EPOLLIN` para detectar nuevas conexiones.
- **Sockets de clientes**: También monitorean `EPOLLIN` para detectar datos entrantes.
- **Simplificación**: Asume que `write` se completa completamente. En producción, búfer de datos y usa

`EPOLLOUT` para escrituras parciales.

- **Manejo de errores**: Cierra sockets en caso de errores o EOF y los elimina de epoll.

Resumen

`epoll` proporciona una manera eficiente de manejar múltiples descriptores de archivos en Linux: 1. Crea una instancia con `epoll_create1`. 2. Registra descriptores de archivos y eventos con `epoll_ctl`. 3. Espera eventos con `epoll_wait`. 4. Procesa eventos en un bucle, ajustando eventos monitoreados o eliminando descriptores según sea necesario.

Para aplicaciones simples, se recomienda el modo disparado por nivel. Para necesidades de alto rendimiento, considera el modo disparado por borde con un manejo cuidadoso de todos los datos disponibles. Siempre empareja `epoll` con E/S no bloqueante para obtener los mejores resultados.