Utilisation d'epoll sous Linux

`epoll` est un appel système Linux conçu pour surveiller efficacement plusieurs descripteurs de fichiers pour des événements, tels que lorsque des données sont disponibles à la lecture ou qu'un socket est prêt à écrire. Il est particulièrement utile en programmation réseau, où un serveur pourrait avoir besoin de gérer de nombreuses connexions simultanément. Comparé à des mécanismes plus anciens comme `select` ou `poll`, `epoll` s'adapte mieux à un grand nombre de descripteurs de fichiers car il évite de scanner à nouveau tous les descripteurs pour la disponibilité.

Voici un guide étape par étape pour utiliser `epoll` sous Linux, suivi d'un exemple simple d'un serveur qui accepte des connexions et renvoie les données.

Étapes pour utiliser `epoll`

1. Créer une instance epoll

Pour commencer à utiliser `epoll`, vous devez créer une instance `epoll`, représentée par un descripteur de fichier. Utilisez l'appel système `epoll_create1` :

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

- **Paramètre** : Passez `0` pour une utilisation de base (aucun drapeau spécial). Des drapeaux comme `EPOLL_CLOEXEC` peuvent être utilisés pour des scénarios avancés.
- **Valeur de retour** : Retourne un descripteur de fichier (`epoll_fd`) en cas de succès, ou `-1` en cas d'erreur (vérifiez `errno` pour plus de détails).

La fonction plus ancienne `epoll_create` est similaire mais prend une indication de taille (maintenant ignorée), donc `epoll_create1` est préférée.

2. Ajouter des descripteurs de fichiers à surveiller

Utilisez `epoll_ctl` pour enregistrer des descripteurs de fichiers (par exemple, des sockets) avec l'instance `epoll` et spécifiez les événements que vous souhaitez surveiller :

```
struct epoll_event ev;
ev.events = EPOLLIN;  // Surveiller la lisibilité
ev.data.fd = some_fd; // Descripteur de fichier à surveiller
epoll_ctl(epoll_fd, EPOLL_CTL_ADD, some_fd, &ev);
```

- **Paramètres** :

  - `epoll_fd` : Le descripteur de fichier de l'instance `epoll`.
  - `EPOLL_CTL_ADD` : Opération pour ajouter un descripteur de fichier.
  - `some_fd` : Le descripteur de fichier à surveiller (par exemple, un socket).
  - `&ev` : Pointeur vers une `struct epoll_event` définissant les événements et les données utilisateur optionnelles.
- **Événements courants** :
  - `EPOLLIN` : Données disponibles à lire.
  - `EPOLLOUT` : Prêt à écrire.
  - `EPOLLERR` : Une erreur s'est produite.
  - `EPOLLHUP` : Déconnexion (par exemple, connexion fermée).
- **Données utilisateur** : Le champ `data` dans `struct epoll_event` peut stocker un descripteur de fichier (comme montré) ou d'autres données (par exemple, un pointeur) pour identifier la source lorsque des événements se produisent.

3. Attendre les événements

Utilisez `epoll_wait` pour bloquer et attendre les événements sur les descripteurs de fichiers surveillés :

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

- **Paramètres** :
  - `epoll_fd` : L'instance `epoll`.
  - `events` : Tableau pour stocker les événements déclenchés.
  - `MAX_EVENTS` : Nombre maximum d'événements à retourner (taille du tableau).
  - `-1` : Délai d'attente en millisecondes (`-1` signifie attendre indéfiniment ; `0` retourne immédiatement).
- **Valeur de retour** : Nombre de descripteurs de fichiers avec événements (`nfds`), ou `-1` en cas d'erreur.

4. Gérer les événements

Parcourez les événements retournés par `epoll_wait` et traitez-les :

```
for (int i = 0; i < nfds; i++) {
    if (events[i].events & EPOLLIN) {
        // Le descripteur de fichier events[i].data.fd est lisible
    }
}
```

- Vérifiez le champ `events` en utilisant des opérations binaires (par exemple, `events[i].events & EPOLLIN`) pour déterminer le type d'événement.
- Utilisez `events[i].data.fd` pour identifier quel descripteur de fichier a déclenché l'événement.

5. Gérer les descripteurs de fichiers (optionnel)

- **Supprimer** : Utilisez `epoll_ctl` avec `EPOLL_CTL_DEL` pour arrêter de surveiller un descripteur de fichier :

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

- **Modifier** : Ajustez les événements avec `EPOLL_CTL_MOD` :

```
ev.events = EPOLLOUT; // Changer pour surveiller la capacité d'écriture
epoll_ctl(epoll_fd, EPOLL_CTL_MOD, some_fd, &ev);
```

Concepts clés

Déclenchement par niveau vs. Déclenchement par bord

- **Déclenchement par niveau (par défaut)** : `epoll` notifie de manière répétée tant que la condition persiste (par exemple, les données restent non lues). Plus simple pour la plupart des cas.
- **Déclenchement par bord (`EPOLLET`)** : Notifie uniquement une fois lorsque l'état change (par exemple, de nouvelles données arrivent). Nécessite la lecture/écriture de toutes les données jusqu'à `EAGAIN` pour éviter de manquer des événements ; plus efficace mais plus délicat.
- Définissez `EPOLLET` dans `ev.events` (par exemple, `EPOLLIN | EPOLLET`) si vous utilisez le mode déclencheur par bord.

E/S non bloquante

`epoll` est souvent associé à des descripteurs de fichiers non bloquants pour éviter le blocage sur les opérations d'E/S. Définissez un socket en mode non bloquant avec :

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

Exemple : Serveur Echo Simple

Voici un exemple de base d'un serveur qui utilise `epoll` pour accepter des connexions et renvoyer les données aux clients. Il utilise le mode déclencheur par niveau pour la simplicité.

```c
#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() {
    // Créer un socket d'écoute
    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); }

    // Définir le socket d'écoute en mode non bloquant
    fcntl(listen_fd, F_SETFL, fcntl(listen_fd, F_GETFL) | O_NONBLOCK);

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

    // Ajouter le socket d'écoute à epoll
    struct epoll_event ev, events[MAX_EVENTS];
    ev.events = EPOLLIN; // Déclencheur par niveau
    ev.data.fd = listen_fd;
    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, listen_fd, &ev) == -1) { perror("epoll_ctl"); exit(1); }

    // Boucle d'événements
    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) {
            // Accepter une nouvelle connexion
            int client_fd = accept(listen_fd, NULL, NULL);
            if (client_fd == -1) { perror("accept"); continue; }

            // Définir le socket client en mode non bloquant
            fcntl(client_fd, F_SETFL, fcntl(client_fd, F_GETFL) | O_NONBLOCK);

            // Ajouter le socket client à 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 {
            // Traiter les données client
            char buf[1024];
            ssize_t n = read(fd, buf, sizeof(buf));
            if (n <= 0) {
                // Erreur ou EOF : fermer et supprimer de epoll
                if (n == -1 && (errno == EAGAIN || errno == EWOULDBLOCK)) continue;
                close(fd);
                epoll_ctl(epoll_fd, EPOLL_CTL_DEL, fd, NULL);
            } else {
                // Renvoi des données
                write(fd, buf, n); // Note : Dans les applications réelles, gérer les écritures partielles
            }
        }
    }
}

close(epoll_fd);
close(listen_fd);
```

```
    return 0;
}
```

Notes sur l'exemple

- **Socket d'écoute** : Surveille `EPOLLIN` pour détecter de nouvelles connexions.
- **Sockets clients** : Surveillent également `EPOLLIN` pour détecter les données entrantes.
- **Simplification** : Suppose que `write` se termine complètement. Dans une application de production, mettez en tampon les données et utilisez `EPOLLOUT` pour les écritures partielles.
- **Gestion des erreurs** : Ferme les sockets en cas d'erreur ou de EOF et les supprime de `epoll`.

Résumé

`epoll` fournit un moyen efficace de gérer plusieurs descripteurs de fichiers sous Linux : 1. Créez une instance avec `epoll_create1`. 2. Enregistrez les descripteurs de fichiers et les événements avec `epoll_ctl`. 3. Attendez les événements avec `epoll_wait`. 4. Traitez les événements dans une boucle, ajustant les événements surveillés ou supprimant les descripteurs selon les besoins.

Pour les applications simples, le mode déclencheur par niveau est recommandé. Pour les besoins de haute performance, envisagez le mode déclencheur par bord avec une gestion attentive de toutes les données disponibles. Associez toujours `epoll` à une E/S non bloquante pour obtenir les meilleurs résultats.