Laboratorio di Reti e Sistemi Distribuiti

6: I/O Multiplexing con epoll()

Roberto Marino, PhD[1]
roberto.marino@unime.it

[1]Dipartimento di Matematica, Informatica, Fisica e Scienze della Terra
Future Computing Research Laboratory
Università di Messina

Last Update: 6th March 2025

Recap: I/O Multiplexing

Definizione

Tecnica che permette a un singolo thread di monitorare multipli file descriptor per:

- Lettura
- Scrittura
- Eccezioni

Perchè è importante?

- Evitare busy waiting
- Gestire connessioni concorrenti usando un singolo thread
- Ottimizzare l'utilizzo delle risorse

Recap: select()

I limiti di select()

Abbiamo visto che la funzione `select()` sposta il monitoraggio dei FD da user-space a kernel-space. Ogni volta che si chiama select, il kernel **itera su tutti gli FD nel set per verificare lo stato**. Questo ha complessità O(n), quindi diventa lento con molti fd. Inoltre, c'è un limite al numero massimo di fd, di solito 1024.

La funzione di epoll()

La funzione epoll() **in Linux** è progettata per superare i limiti di select() e per monitorare efficientemente eventi di I/O su un gran numero di file descriptor (FD). A basso livello, funziona attraverso tre componenti principali: **strutture dati kernel, notifiche asincrone e ottimizzazioni per scalabilità**.

Il funzionamento di epoll()

epoll() usa una struttura dati più efficiente. Con `epoll_create` si crea un'istanza, e con `epoll_ctl` si aggiungono/modificano fd. Il kernel mantiene una lista di interesse e una lista pronta. Quando chiami `epoll_wait`, il kernel restituisce gli fd pronti, senza doverli iterare tutti (perchè li conosce già). La complessità è O(1) per il retrieval di fd pronti, quindi è più scalabile.

Strutture dati internet al kernel

Il kernel gestisce epoll() grazie a due strutture dati chiave:

1. **Interest List (Tree):** Un albero che tiene traccia di tutti i FD registrati con `epoll_ctl()`.
   Ricerca: Efficiente grazie alla struttura ad albero.
2. **Ready List (Linked List):** Una lista collegata che mantiene sempre disponibile i FD pronti per operazioni di I/O e che viene aggiornata dal kernel quando si verificano eventi (es: dati in arrivo su un socket).

La Ready List è acceduta in O(1) durante `epoll_wait()`

epoll() workflow

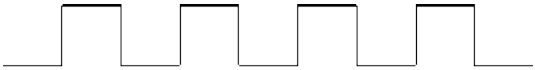
1. **Fase 1**: Registrazione FD (`epoll_ctl`). Quando un FD viene aggiunto a epoll con `epoll_ctl()`, il kernel: Inserisce il FD nell'Interest List. Collega una callback al FD per tracciare eventi (es: dati in arrivo).
2. **Fase 2**: Attesa Eventi (`epoll_wait`). Quando viene chiamato `epoll_wait()`: Il processo viene sospeso finché non ci sono eventi o scade il timeout. Il kernel verifica la Ready List e copia gli eventi pronti nello spazio utente. La Ready List viene svuotata dopo la copia (in modalità edge-triggered).
3. **Fase 3**: Notifica Eventi. Quando un evento si verifica su un FD (es: dati ricevuti): La callback associata al FD viene attivata. Il FD viene aggiunto alla Ready List. Il processo in `epoll_wait()` viene risvegliato (se bloccato).

Modalità di notificà

`epoll()` supporta due modalità:

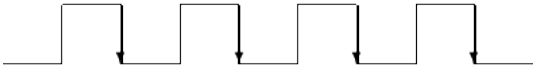
- **Level-Triggered (LT):** Notifica finché il FD è pronto (es: dati non letti nel buffer, finchè presenti).
- **Edge-Triggered (ET):** Notifica solo al cambiamento di stato (es: da "nessun dato" a "dati disponibili").

Edge Triggered vs Level Triggered



(a) Level trigger.

(b) Positive-edge trigger.



(c) Negative-edge trigger.

Time ⟶

Suggerimento per imparare a programmare in Linux

epoll() API

Le tre funzioni fondamentali sono:

- `epoll_create1()` - Crea un'istanza epoll
- `epoll_ctl()` - Gestisce la registrazione dei file descriptor
- `epoll_wait()` - Attende gli eventi

epoll() API

Questa funzione ritorna un nuovo file descriptor per il meccanismo di epoll.

```
1 #include <sys/epoll.h>
2 int epoll_create1(int flags);
```

- `flags`:
  - 0: Comportamento standard
  - `EPOLL_CLOEXEC`: Chiude il FD automaticamente durante exec()

epoll() API

```
1 int epoll_fd = epoll_create1(0);
2 if (epoll_fd == -1) {
3     perror("epoll_create1");
4     exit(EXIT_FAILURE);
5 }
```

epoll_ctl() API

Questa funzione aggiunge, modifica o rimuove un file descriptor gestito dall'epoll.

```
1 #include <sys/epoll.h>
2 int epoll_ctl(int epoll_fd, int op, int fd, struct epoll_event *event)
    ;
```

- `epoll_fd`: Descrittore restituito da epoll_create1
- `op`: Operazione da eseguire:
  - `EPOLL_CTL_ADD`: Aggiunge fd alla lista degli eventi ì monitorati con i settaggi specificati in event.
  - `EPOLL_CTL_MOD`: Modifica le condizioni di monitoraggio di fd.
  - `EPOLL_CTL_DEL`: Rimuove fd dalla lista.
- `fd`: File descriptor da gestire
- `event`: Struttura di configurazione

Esempio di utilizzo di epoll_ctl

```
1 struct epoll_event event;
2 event.events = EPOLLIN | EPOLLET;
3 event.data.fd = sockfd;
4
5 if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, sockfd, &event) == -1) {
6     perror("epoll_ctl");
7     exit(EXIT_FAILURE);
8 }
```

`struct epoll_event` Definisce un evento monitorato da epoll e contiene due campi principali:

- **events:** Specifica quali eventi devono essere monitorati.
- **data:** Contiene dati personalizzati che l'utente può associare all'evento.

Set per FD

Lista non esaustiva dei set possibili:

- **EPOLLIN**: Il file descriptor è pronto per la lettura.
- **EPOLLOUT**: Il file descriptor è pronto per la scrittura.
- **EPOLLET**: Modalità Edge-Triggered (Default: level triggered)

Strutture dati

```
1  struct epoll_event {
2      uint32_t events;
3      epoll_data_t data;
4  };
5
6  typedef union epoll_data {
7      void *ptr;
8      int fd;
9      uint32_t u32;
10     uint64_t u64;
11 } epoll_data_t;
```

epoll_wait()

```
1 #include <sys/epoll.h>
2 int epoll_wait(int epoll_fd, struct epoll_event *events,
3                int maxevents, int timeout);
```

- `epoll_fd`: Descrittore epoll
- `events`: Un array di struct `epoll_event` per memorizzare gli eventi rilevati.
- `maxevents`: Capacità del buffer
- `timeout`: Timeout in millisecondi (-1 = infinito)

epoll_wait()

Questa funzione attende eventi sui file descriptor monitorati da epoll.

```c
#define MAX_EVENTS 10
struct epoll_event events[MAX_EVENTS];

int num_events = epoll_wait(epoll_fd, events, MAX_EVENTS, -1);
if (num_events == -1) {
    perror("epoll_wait");
    exit(EXIT_FAILURE);
}

for (int i = 0; i < num_events; i++) {
    // Gestione eventi
}
```