Universita' di Messina — CdL in Informatica — Reti e Sistemi Distribuiti Mod.B

Hands-On #6

I/O multiplexing con `epoll()`: O(1) per connessione

Last Update: March 19, 2026

Scenario

Nell'HO#5 hai trasformato il server mono-client dell'HO#1 in un server multi-client usando `select()`. Funziona, ma ha un limite strutturale: ad ogni ciclo del loop il kernel scansiona **tutti** gli fd monitorati, anche quelli idle. Con 10.000 client e 3 attivi, esegue 9.997 controlli inutili ad ogni iterazione. La complessita' e' O(n) per ogni chiamata a `select()`.

`epoll()` risolve questo alla radice. Il kernel mantiene internamente un **albero rosso-nero** con tutti gli fd registrati e una **ready list** con i soli fd che hanno eventi attivi. `epoll_wait()` non scansiona nulla: consegna direttamente cio' che e' nella ready list. Il costo scende a **O(eventi pronti)**, indipendente dal numero totale di fd.

Questo e' il motivo per cui Nginx, Redis e Node.js usano `epoll()` come motore I/O su Linux.

Concetti chiave da tenere a mente

Le tre syscall di `epoll()`

`epoll()` non e' una singola chiamata come `select()`, ma un'interfaccia composta da tre syscall con ruoli distinti.

`epoll_create1(0)` crea un'istanza epoll nel kernel e restituisce un file descriptor che la rappresenta (`epfd`). Si chiama **una volta sola** prima del loop.

`epoll_ctl()` registra, modifica o rimuove un fd dall'istanza epoll. Si chiama **una volta per fd**, solo quando un client si connette o si disconnette — mai dentro il loop critico. Il costo e' O(log n) sull'albero rosso-nero interno.

`epoll_wait()` attende eventi e restituisce *solo* gli fd pronti in un array pre-allocato. Non riceve nessun set da costruire, non copia nulla. Il costo e' O(eventi pronti).

Listing 1: Le tre syscall — struttura

```c
/* 1. Una volta sola */
int epfd = epoll_create1(0);

/* 2. Per ogni nuovo client (e alla disconnessione) */
struct epoll_event ev;
ev.events  = EPOLLIN;
ev.data.fd = client_fd;
epoll_ctl(epfd, EPOLL_CTL_ADD, client_fd, &ev);

/* 3. Ad ogni ciclo del loop */
struct epoll_event events[MAX_EVENTS];
int n = epoll_wait(epfd, events, MAX_EVENTS, -1);
for (int i = 0; i < n; i++) {
    int fd = events[i].data.fd;  /* fd pronto */
    recv(fd, buf, sizeof(buf), 0);
```

```
}
```

Confronto diretto con `select()`

Aspetto	select()	epoll()
Set da ricostruire	Si', ad ogni ciclo	No, mai
Copia kernel/ciclo	O(n) — tutto il set	Nessuna
Attesa eventi	O(n) — scansione totale	O(eventi pronti)
Registrazione fd	O(n)/ciclo	O(log n) una volta
Limite fd	~1024	Nessuno
`max_fd`	Da tenere aggiornato	Non esiste

⋆ Perche' non serve max_fd con epoll

Con `select()` il primo argomento `nfds = max_fd + 1` indica al kernel fino a quale fd scansionare. Con `epoll()` il kernel non scansiona nulla: conosce gia' i fd registrati dall'albero rosso-nero. Il concetto di "fd massimo" non esiste.

Le tre operazioni di `epoll_ctl()`

Operazione	Quando si usa	Costo
`EPOLL_CTL_ADD`	Nuovo client accettato	O(log n)
`EPOLL_CTL_MOD`	Cambio eventi monitorati	O(log n)
`EPOLL_CTL_DEL`	Client disconnesso	O(log n)

[!] Prima DEL, poi close()

Rimuovi sempre il fd da epoll con `EPOLL_CTL_DEL` **prima** di chiamare `close(fd)`. Chiudere un fd senza rimuoverlo da epoll causa comportamento indefinito: su kernel recenti il fd viene rimosso automaticamente alla chiusura, ma fare affidamento su questo e' una cattiva pratica.

Codice da analizzare: `server_epoll.c`

Ti viene fornito il file `server_epoll.c`: un server multi-client funzionante che usa `epoll()` in modalita' level-triggered. E' l'equivalente diretto di `server_select_demo.c` dell'HO#5 — stessa funzionalita', implementazione diversa.

Leggi il codice con attenzione prima di procedere, tenendo a mente queste domande guida:

- Dove viene chiamata `epoll_create1()`? Una volta o ad ogni ciclo?
- Dove viene chiamata `epoll_ctl(ADD)`? Quante volte per ogni client?
- Dove viene chiamata `epoll_ctl(DEL)`? Prima o dopo `close()`?
- Dove si trova il loop che in `select()` faceva `FD_ZERO` + `FD_SET`? Esiste ancora?
- Dov'e' `max_fd`? Esiste ancora?

Listing 2: Struttura centrale di server_epoll.c

```
/* 1. Una volta sola: crea l'istanza epoll */
int epfd = epoll_create1(0);

```

```c
/* 2. Una volta sola: registra server_fd */
ev.events = EPOLLIN; ev.data.fd = server_fd;
epoll_ctl(epfd, EPOLL_CTL_ADD, server_fd, &ev);

while (1) {
    /* 3. Attende eventi: NO set, NO max_fd, NO copia */
    int n = epoll_wait(epfd, events, MAX_EVENTS, -1);

    for (int i = 0; i < n; i++) {
        int fd = events[i].data.fd;

        if (fd == server_fd) {
            /* nuovo client: accept + epoll_ctl(ADD) */
            int cfd = accept(server_fd, NULL, NULL);
            ev.events = EPOLLIN; ev.data.fd = cfd;
            epoll_ctl(epfd, EPOLL_CTL_ADD, cfd, &ev);

        } else {
            ssize_t r = recv(fd, buf, sizeof(buf)-1, 0);
            if (r <= 0) {
                /* disconnessione: DEL poi close */
                epoll_ctl(epfd, EPOLL_CTL_DEL, fd, NULL);
                close(fd);
            } else { /* elabora buf[0..r-1] */ }
        }
    }
}
```

Esercizi

Esercizio 1 — Analisi di `server_epoll.c`

Compila ed esegui `server_epoll.c`. Poi rispondi alle seguenti domande *guardando il codice e osservando l'output*:

1. `server_epoll.c` e `server_select_demo.c` producono lo stesso output quando si connettono tre client. Qual e' la differenza strutturale piu' evidente nel codice? Quante righe sono cambiate?
2. In `server_select_demo.c` esistevano `client_fds[]` e `max_fd`. In `server_epoll.c` dove sono finite queste strutture? Chi si occupa ora di tenere traccia dei fd attivi?
3. Connetti cinque client con lo script:

   ```
   ./test_multifd.sh 5
   ```

   Osserva la riga `epoll_wait() ritorna: N fd pronti`. Il valore di N e' sempre 1 oppure a volte e' maggiore? Perche'?
4. `epoll_ctl(DEL)` viene chiamato **prima** di `close(fd)` o dopo? Perche' e' importante questo ordine?
5. `epoll_create1()` viene chiamato una volta sola prima del loop. Cosa succederebbe se lo chiamassi ad ogni iterazione? Quale risorsa verrebbe sprecata?
6. Confronta la riga di compilazione di `server_epoll.c` con quella di un programma che usa `io_uring`. Quale libreria aggiuntiva richiederebbe `io_uring` e perche' `epoll()` non ne ha bisogno?

▷ **Per osservare piu' fd pronti**

Usa lo script con un numero alto di client e un delay basso:

```
./test_multifd.sh 10 0
```

Il secondo parametro `0` riduce il delay prima dell'invio, aumentando la probabilita' che piu' messaggi arrivino contemporaneamente nello stesso ciclo di `epoll_wait()`.

Esercizio 2 — Modifica `server_tcp.c` con `epoll()`

Parti da `server_tcp.c` dell'HO#1 e trasformalo in un server multi-client usando `epoll()`, in modo che funzioni correttamente con `test_multifd.sh`.
Il tuo obiettivo e' che questo comando:

```
./test_multifd.sh 5
```

produca sull'output del server cinque messaggi ricevuti da cinque fd diversi, senza blocchi e senza perdite.

Le modifiche necessarie sono **quattro**, ciascuna in un punto preciso del codice:

1. **Aggiungi l'include e la costante**

```c
#include <sys/epoll.h>
#define MAX_EVENTS 64
```

2. **Crea l'istanza epoll e registra `server_fd`** (subito dopo `listen()`, prima del loop):

```c
int epfd = epoll_create1(0);
struct epoll_event ev, events[MAX_EVENTS];
ev.events = EPOLLIN; ev.data.fd = server_fd;
epoll_ctl(epfd, EPOLL_CTL_ADD, server_fd, &ev);
```

3. **Sostituisci `accept()` fisso con `epoll_wait()`:**

```c
/* PRIMA (HO#1): */
client_fd = accept(server_fd, NULL, NULL);

/* DOPO: */
int n = epoll_wait(epfd, events, MAX_EVENTS, -1);
for (int i = 0; i < n; i++) {
    int fd = events[i].data.fd;
    if (fd == server_fd) {
        int cfd = accept(server_fd, NULL, NULL);
        ev.events = EPOLLIN; ev.data.fd = cfd;
        epoll_ctl(epfd, EPOLL_CTL_ADD, cfd, &ev);
    } else {
        /* gestisci dati --- passo 4 */
    }
}
```

4. **Sostituisci il loop `recv()` con la gestione nel ramo `else`:**

```c
/* PRIMA (HO#1): */
while (1) {
    n = recv(client_fd, buf, BUFLEN, 0);
    if (n <= 0) break;
    printf("%s\n", buf);
}
```

```
7
8  /* DOPO (dentro il ramo else): */
9  ssize_t r = recv(fd, buf, BUFLEN - 1, 0);
10 if (r <= 0) {
11     epoll_ctl(epfd, EPOLL_CTL_DEL, fd, NULL);
12     close(fd);
13 } else {
14     buf[r] = '\0';
15     printf("[fd=%d] %s\n", fd, buf);
16 }
```

Verifica il funzionamento con:

- `./test_multifd.sh 1` — un solo client: comportamento identico al server originale dell'HO#1
- `./test_multifd.sh 5` — cinque client simultanei: tutti i messaggi devono arrivare
- `./test_multifd.sh 10` — dieci client: nessun crash, nessuna perdita

[!] Aggiungi SO_REUSEADDR e set_nonblocking

Aggiungi `SO_REUSEADDR` subito dopo `socket()` per riavviare il server immediatamente:

```
1 int opt = 1;
2 setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR,
3             &opt, sizeof(opt));
```

E rendi non bloccante `server_fd` con `fcntl()` prima di registrarlo in epoll (vedi `server_epoll.c` per il pattern esatto).

[!] Non dimenticare `close(epfd)`

Prima di terminare il programma chiudi l'istanza epoll:

```
1 close(epfd);
2 close(server_fd);
```

`epfd` e' un file descriptor come gli altri: se non lo chiudi il kernel mantiene allocata la struttura dati interna.

File da consegnare: `server_tcp_epoll.c` e report PDF con le risposte alle domande dell'Esercizio 1.

Consegna: inviare zip al docente (`roberto.marino@unime.it`)

Prossimo passo: `epoll()` e' la base di tutti i server ad alte prestazioni su Linux, ma e' ancora un modello *readiness* — ti dice quando puoi leggere, ma sei tu a leggere. Il passo successivo e' `io_uring`: modello *completion*, il kernel legge per te e ti consegna i dati gia' pronti, con zero syscall nel caso ottimale.