I/O Multiplexing

Socket bloccanti, non bloccanti e la funzione `select()`

Roberto Marino, PhD

Università degli studi di Messina
CdL Informatica — Reti e Sistemi Distribuiti Mod.B

Last Update: March 19, 2026



UniMe
1548

Il server che conosciamo già

Listing 1: server bloccante iterativo

```
1  server_fd = socket(...);
2  bind(...); listen(...);
3
4  /* blocca qui finche' arriva
5     una connessione */
6  client_fd = accept(server_fd,
7                     NULL, NULL);
8
9  while (1) {
10   /* blocca qui finche' arrivano
11      dati dal client */
12   n = recv(client_fd, buf,
13            BUFLEN, 0);
14   if (n <= 0) break;
15   printf("%s\n", buf);
16 }
```

Problema architetturale

Un solo client alla volta.

Mentre `recv()` aspetta dati dal client A, il client B resta in coda e non viene mai servito.

Domanda

Come gestiamo **N client** contemporaneamente con un solo processo?

Due modalità operative

[B] Socket **bloccante** (default)

- `recv()` → sospende il processo se non ci sono dati
- `accept()` → sospende se non ci sono connessioni
- `send()` → sospende se il buffer è pieno
- Semplice da usare, ma un solo cliente per volta

[NB] Socket **non bloccante**

- `recv()` → ritorna **subito** con `-1/EAGAIN` se non ci sono dati
- `accept()` → ritorna `-1/EAGAIN` se nessuno aspetta
- `send()` → ritorna `-1/EAGAIN` se buffer pieno
- Richiede gestione esplicita degli errori

Come rendere un socket non bloccante

Listing 2: Impostare O_NONBLOCK con fcntl()

```c
#include <fcntl.h>

/* Passo 1: leggo i flag attuali */
int flags = fcntl(sockfd, F_GETFL, 0);
if (flags == -1) {
    perror("fcntl F_GETFL");
    exit(1);
}

/* Passo 2: aggiungo O_NONBLOCK senza toccare gli altri flag */
if (fcntl(sockfd, F_SETFL, flags | O_NONBLOCK) == -1) {
    perror("fcntl F_SETFL");
    exit(1);
}
/* Da questo momento recv(), send(), accept() non bloccano mai */
```

Come rendere un socket non bloccante

Codici di errore da gestire (man errno

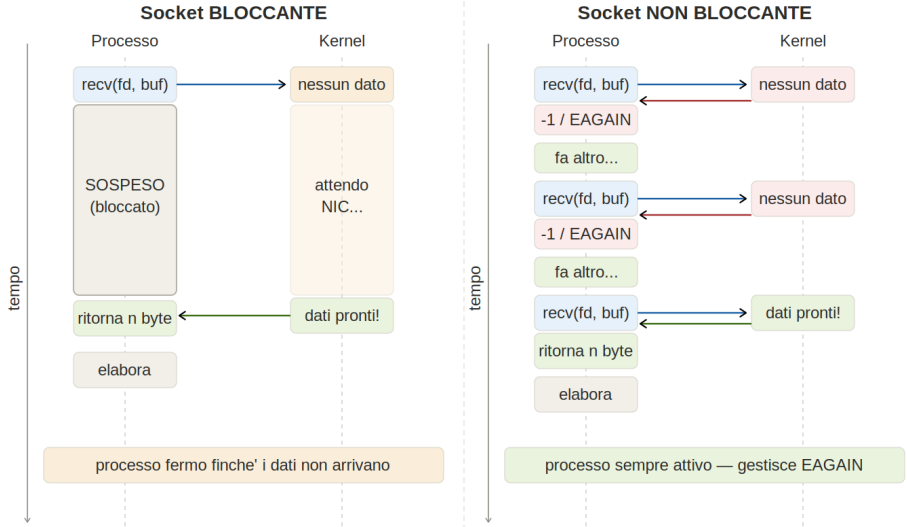
- `EAGAIN` — nessun dato disponibile, significa letteralmente "NON CI SONO DATI! PROVA ANCORA...!"
- `EWOULDBLOCK` — sinonimo di `EAGAIN` su molti sistemi
- Controlla **sempre** `errno` dopo `-1`

[!] Attenzione: busy-waiting

Loop che chiama `recv()` in continuazione senza aspettare consuma il **100% della CPU** anche quando non c'è nulla da fare.

$\Rightarrow$ Serve un meccanismo di **attesa efficiente**.

Blocking vs non-blocking

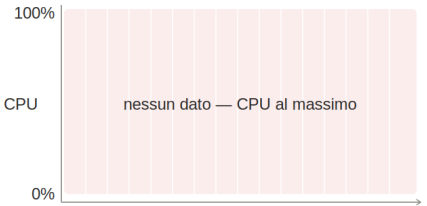


Il problema del busy-waiting

Busy-waiting — SBAGLIATO

100%
CPU sprecata

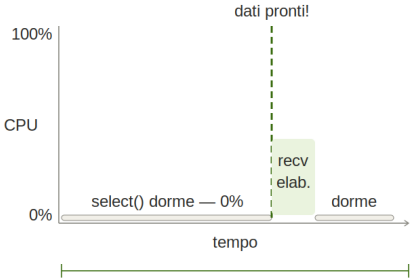
Loop infinito:
while(1) {
recv() → EAGAIN
recv() → EAGAIN
recv() → EAGAIN
}



I/O MULTIPLEXING — CORRETTO

~0%
CPU a riposo

Il kernel avvisa solo
quando ci sono dati:
zero spreco di CPU



Come risolvere il busy waiting?

Principio fondamentale dell'IO multiplexing

Non fare polling. Lasciare che sia il **kernel** ad avvisare quando un socket è pronto — questo è **l'I/O multiplexing**.

La chiamata select()

Listing 3: Prototipo di select() — <sys/select.h>

```
int select(int nfds,     /* da settare alla dimensione pi  grande tra
    readfs, writefds
           fd_set *readfds,   /* fd da monitorare in lettura  */
           fd_set *writefds,  /* fd da monitorare in scrittura */
           fd_set *exceptfds, /* condizioni eccezionali        */
           struct timeval *timeout); /* NULL = attendi forever */

/* Valore di ritorno:
 *  > 0  : numero di fd pronti
 *  = 0  : timeout scaduto, nessun fd pronto
 *  = -1 : errore (controlla errno)
 */
```

La chiamata `select()`

Macro per `fd_set`

`FD_ZERO(&s)`	azzera il set
`FD_SET(fd,&s)`	aggiunge `fd`
`FD_CLR(fd,&s)`	rimuove `fd`
`FD_ISSET(fd,&s)`	testa se pronto

[!] Attenzione!

`select()` **modifica** i set in input!
Dopo la chiamata i set contengono **solo** gli fd pronti.

$\Rightarrow$ Ricostruisci i set ad ogni iterazione del loop.

select() modifica il set — il problema

Prima di chiamare `select()`, costruiamo il set con i fd che vogliamo monitorare:

```
fd_set read_set;
FD_ZERO(&read_set);
FD_SET(4, &read_set); /* client A */
FD_SET(5, &read_set); /* client B */
FD_SET(6, &read_set); /* client C */

/* read_set ora contiene: {4, 5, 6}
    */
```

Attenzione

`select()` usa lo **stesso** set sia come *input* (fd da monitorare) che come *output* (fd pronti).

Sono due cose diverse contenute nella stessa variabile.

select() modifica il set — cosa succede dopo

Arrivano,ad esempio, dati **solo su fd 5** (client B). Il kernel sovrascrive il set rimuovendo i fd *non* pronti:

```
/* Prima della chiamata */
/* read_set = {4, 5, 6}  <- tutti i fd monitorati */

select(7, &read_set, NULL, NULL, NULL);

/* Dopo la chiamata */
/* read_set = {5}          <- solo i fd pronti!     */
/* fd 4 e fd 6 sono stati RIMOSSI dal kernel        */
```

Conseguenza

Al ciclo successivo `read_set` contiene solo `{5}`. Se richiami `select()` senza ricostruire il set, client A e client C non verranno **mai** piu' serviti.

`select()` e' bloccante?

Si' — ma in modo controllato

`select()` sospende il processo se nessun fd nel set ha eventi pronti. Il processo non consuma CPU: il kernel lo mette in stato di attesa (*sleeping*) finche' qualcosa accade.

`recv()` bloccante — blocco individuale

Blocca su **un solo fd**.

Se quel client non manda nulla il processo resta fermo per sempre, e tutti gli altri client vengono ignorati.

`select()` — blocco collettivo

Blocca su **tutti gli fd insieme**.

Appena uno qualsiasi ha un evento, il processo si sveglia e sa esattamente quale fd servire.

Quando `select()` ritorna

`select()` ritorna in tre casi distinti:

1 — Almeno un fd e' pronto (caso normale)

Uno o piu' fd hanno dati disponibili, sono pronti per la scrittura, o hanno un errore. Ritorna il numero di fd pronti e sovrascrive il set con solo quelli attivi.

2 — Timeout scaduto

Se si passa una `struct timeval` invece di `NULL`, `select()` ritorna `0` allo scadere del tempo anche se nessun fd e' pronto. E' il meccanismo degli heartbeat.

3 — Interruzione da segnale

Se il processo riceve un segnale (es. `SIGINT`) mentre e' bloccato, `select()` ritorna `-1` con `errno == EINTR`. Va gestito esplicitamente.

Che cos'e' errno?

Definizione

errno e' una variabile globale definita in <errno.h> che il kernel usa per comunicare al processo il **motivo** per cui una syscall e' fallita.

Quando una funzione di sistema ritorna -1, non basta sapere che qualcosa e' andato storto — serve sapere *cosa* e' andato storto.

```c
#include <errno.h>

int n = recv(sockfd, buf, sizeof(buf), 0);

if (n == -1) {
    /* errno e' stato impostato dal kernel */
    if (errno == EAGAIN)     printf("nessun dato ora\n");
    if (errno == ECONNRESET) printf("connessione resettata\n");
    if (errno == EBADF)      printf("fd non valido\n");
}
```

Gestire i tre valori di ritorno

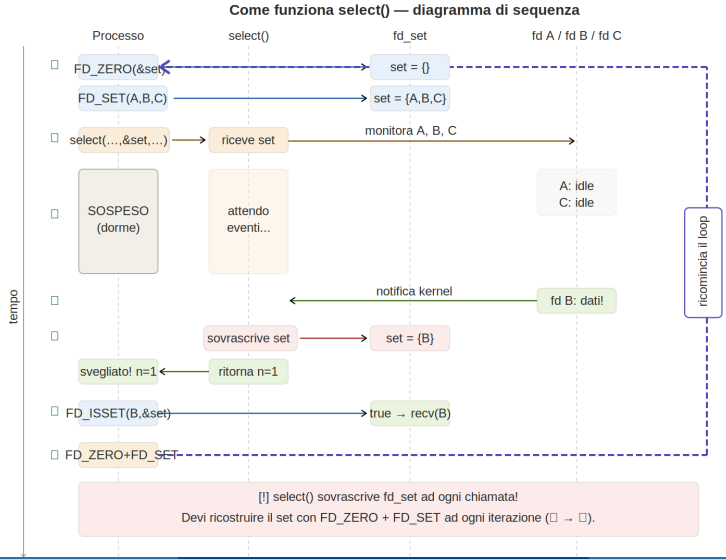
```c
int n = select(max_fd + 1, &read_set,
               NULL, NULL, NULL);

if (n < 0) {
    if (errno == EINTR) continue; /* segnale: riprova */
    perror("select");  break;     /* errore vero      */
}

if (n == 0) {
    /* timeout scaduto: nessun fd pronto */
    /* momento ideale per heartbeat      */
}

if (n > 0) {
    /* n fd sono pronti                  */
    /* controlla uno per uno con FD_ISSET */
}
```

Come funziona `select()` — diagramma



Quanti fd restituisce `select()`?

select() ritorna appena il primo fd e' pronto

Non esiste una finestra temporale. `select()` si sveglia non appena **almeno un** fd ha eventi — ma nel set restituisce **tutti** quelli pronti in quell'istante.

```
1 /* t = 0 ms:    select() si blocca, nessun fd pronto     */
2 /* t = 10 ms:   dati su fd=4 -> kernel sveglia processo */
3 /* t = 10 ms + scheduling: fd=6 e fd=7 nel frattempo    */
4 /*              sono diventati pronti                    */
5
6 int n = select(max_fd+1, &read_set, NULL, NULL, NULL);
7 /* n == 3: fd 4, 6, 7 tutti nel set al momento del      */
8 /*          ritorno, anche se solo fd=4 ha "scatenato"  */
9 /*          il risveglio                                 */
```

Quanti fd restituisce `select()`?

Perche' possono essere piu' di uno

Tra il momento in cui il kernel rileva il primo evento e il momento in cui il processo viene schedulato passano alcuni microsecondi. In quell'intervallo altri fd possono diventare pronti. Il kernel li marca tutti nel set **prima** che il processo lo legga: `select()` restituisce quindi uno *snapshot* di tutto cio' che e' pronto in quell'istante — non solo il primo.

Come si usa FD_ISSET

Definizione

`FD_ISSET(fd, &set)` ritorna un valore **non-zero** se `fd` e' nel set dei pronti dopo il ritorno di `select()`, **zero** altrimenti. Non dice quanti byte ci sono — dice solo che una `recv()` su quel fd non si bloccherà.

Server multi-client con `select()` — il problema

Definizione del problema

Supponiamo adesso che un server debba gestire **N client contemporaneamente** con un solo thread. Non sa quale fd avra' dati per primo: se chiama `recv()` su un client e gli altri mandano dati, il processo resta bloccato e i messaggi non vengono mai letti.

Soluzione: `select()` sospende il processo finche' **almeno un fd e' pronto**, poi il server serve solo quelli attivi.

Server multi-client con `select()`

```c
int client_fds[MAX_CLIENTS]; /* -1 = slot libero */
int max_fd = server_fd;

while (1) {
    fd_set read_set;
    FD_ZERO(&read_set);
    FD_SET(server_fd, &read_set); /* nuove connessioni */

    for (int i = 0; i < MAX_CLIENTS; i++)
        if (client_fds[i] != -1)
            FD_SET(client_fds[i], &read_set);

    select(max_fd + 1, &read_set, NULL, NULL, NULL);
    /* ... gestione eventi nel frame successivo */
}
```

Server multi-client con `select()`

Listing 4: Gestione nuove connessioni e dati in arrivo

```
    /* Nuovo client in arrivo? */
    if (FD_ISSET(server_fd, &read_set)) {
        int new_fd = accept(server_fd, NULL, NULL);
        /* salva new_fd in client_fds[] */
        /* if (new_fd > max_fd) max_fd = new_fd; */
    }

    /* Dati da un client esistente? */
    for (int i = 0; i < MAX_CLIENTS; i++) {
        if (client_fds[i] != -1 &&
            FD_ISSET(client_fds[i], &read_set)) {
            /* leggi dati o chiudi connessione */
        }
    }
```

Server Loop con `select()` — struttura dati

```
1 int client_fds[MAX_CLIENTS]; /* -1 = slot libero */
2 int max_fd = server_fd;
```

`client_fds[]` — la rubrica dei client

Array che tiene traccia di tutti i client connessi. Ogni slot contiene il fd di un client oppure -1 se libero. Non viene mai toccato da `select()`: e' la nostra memoria persistente tra un ciclo e l'altro.

`max_fd` — il fd piu' grande

Serve come primo argomento a `select()`. Va aggiornato ogni volta che si connette un nuovo client con fd numericamente piu' grande del precedente massimo.

Server Loop con `select()` — ricostruzione del set

```c
while (1) {
    fd_set read_set;
    FD_ZERO(&read_set);
    FD_SET(server_fd, &read_set); /* nuove connessioni */

    for (int i = 0; i < MAX_CLIENTS; i++)
        if (client_fds[i] != -1)
            FD_SET(client_fds[i], &read_set);

    select(max_fd + 1, &read_set, NULL, NULL, NULL);
    /* ... */
}
```

Perche' ricostruire il set ad ogni ciclo?

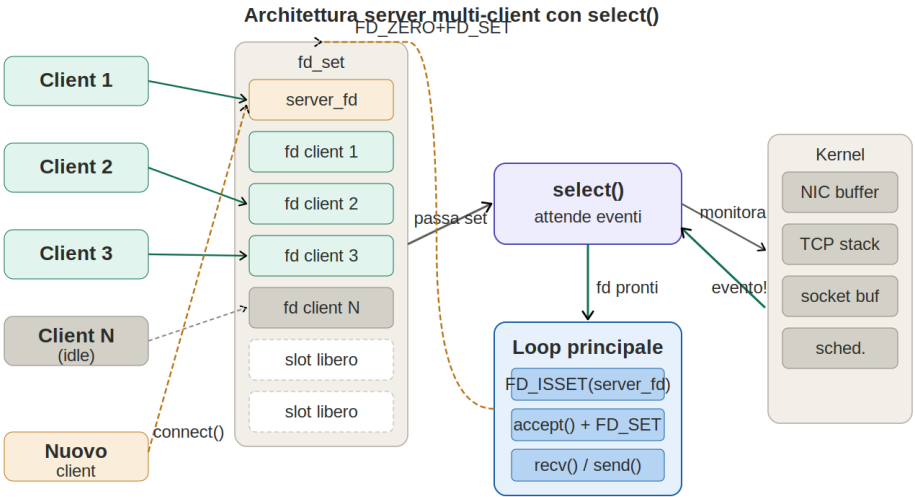
`select()` sovrascrive `read_set` lasciando solo i fd pronti. Al ciclo successivo il set e' corrotto: va azzerato con `FD_ZERO` e ricostruito da `client_fds[]` prima di ogni nuova chiamata.

Server Loop con `select()` — gestione degli eventi

```c
/* Nuovo client in arrivo? */
if (FD_ISSET(server_fd, &read_set)) {
    int new_fd = accept(server_fd, NULL, NULL);
    /* salva new_fd in client_fds[] */
    /* aggiorna max_fd se new_fd > max_fd */
}

/* Dati da un client esistente? */
for (int i = 0; i < MAX_CLIENTS; i++) {
    if (client_fds[i] != -1 &&
        FD_ISSET(client_fds[i], &read_set)) {
        ssize_t n = recv(client_fds[i], buf,
                         sizeof(buf), 0);
        if (n <= 0) client_fds[i] = -1; /* disconnesso */
        else        /* elabora buf[0..n-1] */
    }
}
```

Architettura server con `select()`



I/O multiplexing vs thread per client

[IO] I/O Multiplexing

Pro:

- Un solo thread, nessun context switch
- Memoria minima (no stack per thread)
- Nessuna race condition
- Ottimo per molte connessioni idle

Contro:

- Codice più complesso
- Nessun parallelismo reale su multi-core
- Un'operazione lenta blocca tutto

[T] Thread per client

Pro:

- Codice semplice e lineare
- Parallelismo reale su multi-core
- Una connessione lenta non blocca le altre

Contro:

- Ogni thread $\approx$ 8 MB di stack
- Context switch costoso con molti thread
- Race condition e sincronizzazione
- Limite pratico: $\sim$1000 thread

Soluzione moderna(ibrida): Server come Nginx usano **N worker process**, ognuno con il proprio `epoll`. Parallelismo su core + efficienza dell'I/O multiplexing.

Strumenti diagnostici

Strumenti di debugging utili

- `strace -e trace=network ./server` — traccia le syscall di rete
- `ss -tnp` — mostra connessioni TCP attive con PID
- `htop` — verifica che la CPU sia vicino allo 0% in attesa

Complessita' di `select()` — il costo di attesa

Costo O(n) ad ogni chiamata

Il kernel scansiona **tutti** gli fd nel set da 0 fino a `nfds-1`, anche quelli idle.

Con 10.000 client e 3 attivi: 9.997 controlli inutili ad ogni ciclo.

```
1  /* 3 client attivi su 10.000 */
2  select(10001, &set, NULL, NULL, NULL);
3
4  /* il kernel controlla:
5   *   fd=0   ... fd=3    idle
6   *   fd=4               PRONTO   <- client 1
7   *   fd=5   ... fd=997  idle
8   *   fd=998             PRONTO   <- client 2
9   *   fd=999...fd=9997   idle
10  *   fd=9998            PRONTO   <- client 3
11  *   10.000 iterazioni per soli 3 eventi
```

Complessita' di `select()` — il costo di copia

Costo O(n) di copia ad ogni ciclo

Ad ogni chiamata `select()` copia l'intero `fd_set` da userspace a kernel e lo restituisce modificato.

Il set va poi ricostruito da zero prima del ciclo successivo: tre operazioni O(n) per ogni iterazione del loop.

```
while (1) {
    FD_ZERO(&set);            /* O(n): azzera tutto */
    for (i = 0; i < n; i++)
        FD_SET(fds[i], &set); /* O(n): ricostruisce */

    select(max+1, &set, ...); /* O(n): copia + scansione */

    /* Al prossimo ciclo: ricomincia da capo */
}
```

Complessita' di `select()` — il limite FD_SETSIZE

Limite strutturale: FD_SETSIZE ≈ 1024

`fd_set` e' internamente una bitmask di dimensione fissa definita a compile time.

Non e' possibile monitorare fd con valore superiore a `FD_SETSIZE` senza ricompilare il kernel. Su sistemi moderni con migliaia di connessioni questo e' un limite invalicabile.

```
/* fd_set e' una bitmask: un bit per ogni fd */
/* FD_SETSIZE e' tipicamente 1024             */

FD_SET(1023, &set);  /* OK  */
FD_SET(1024, &set);  /* undefined behavior! */

/* con accept() il kernel assegna fd in ordine
 * crescente: oltre 1024 client, select() non
 * puo' monitorarli                          */
```

Riferimenti

Libri consigliati

- **Stevens, Fenner, Rudoff**
  Unix Network Programming, Vol.1
  Cap. 6 (I/O Multiplexing)
- **Kerrisk**
  The Linux Programming Interface
  Cap. 63 (epoll)
- **Beej's Guide to Network Programming**
  `beej.us/guide/bgnet`

Man pages fondamentali

- `man 2 select`
- `man 2 fcntl`
- `man 7 socket`

Domande?

`roberto.marino@unime.it`