Laboratorio di Reti e Sistemi Distribuiti

4: Socket Bloccanti e Non-Bloccanti

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: 4th March 2025

Chiamate bloccanti

Definizione

Una **chiamata bloccante** è una funzione o un'operazione che sospende l'esecuzione del programma finché non viene completata. Durante l'attesa, il thread o il processo che ha invocato la chiamata rimane inattivo ("dormiente"), senza consumare cicli di CPU, finché l'operazione non termina.

Come funzionano?

1. **Invocazione:** Il programma chiama una funzione bloccante (es. `accept()`, `read()`, `connect()`).
2. **Attesa:**
   - Se la risorsa non è pronta (es: nessun client in connessione, dati non disponibili), il sistema operativo sospende il thread ed esegue **un programma differente**
   - Il controllo della CPU passa ad altri processi/thread.
3. **Ripresa:**
   - Quando l'evento atteso si verifica (es: arriva una connessione), il sistema operativo riattiva il thread.
   - La funzione bloccante ritorna, e il programma riprende l'esecuzione.

Vantaggi e svantaggi di una blocked call

Vantaggi

- Semplicità: Il codice è lineare e facile da seguire.
- Efficienza: Zero consumo di CPU durante l'attesa (no busy-waiting).

Svantaggi

- Scalabilità limitata:
  - In un server bloccante, ogni client deve attendere il completamento delle operazioni del client precedente.
  - Esempio: Se il Client A è in attesa di dati, il server non può gestire il Client B finché Client A non ha terminato.
- Risposta lenta: Inapplicabile per scenari real-time con molte richieste concorrenti.

Esempi di chiamate bloccanti

```
1 int client_socket = accept(server_socket, ...);
```

Il **server** si blocca qui finchè un client non si connette, e non consuma CPU (context-switch!)

```
1 char buffer[256];
2 int bytes = read(client_socket, buffer, sizeof(buffer));
```

Il **server** si blocca qui finchè un client non invia dati, e non consuma CPU (context-switch!)

```
1 connect(socket, (struct sockaddr*)&address, sizeof(address));
```

Il **client** si blocca finchè la connessione non viente stabilita, e non consuma CPU (context-switch!)

Esempi di codice

Codice Bloccante (Sincrono)

```
// Si blocca finch  non arriva una connessione
int client_socket = accept(server_socket, ...);
```

Codice Non-Bloccante (Pseudo-Asincrono con polling)

```
// Configura il socket come non-bloccante
fcntl(server_socket, F_SETFL, O_NONBLOCK);

while(1) {
    int client_socket = accept(server_socket, ...);
    if (client_socket == -1 && errno == EWOULDBLOCK) {
        // Nessuna connessione: continua a provare
        continue;
    }
    // Gestisci il client...
}
```

La funzione fcntl()

`fcntl()` (abbreviazione di **"file control"**) è una system call Unix/Linux per manipolare i descrittori di file (file descriptor). Permette di modificare le proprietà di un file o socket aperto, attraverso il suo file descriptor.

```c
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* arg */ );
```

```c
// Ottieni i flag correnti
int flags = fcntl(fd, F_GETFL);

// Aggiungi il flag O_NONBLOCK (non-bloccante)
fcntl(fd, F_SETFL, flags | O_NONBLOCK);
```

Server Bloccante (senza Attesa Attiva)

```
while (1) {
        // Accetta una connessione (bloccante)
        if ((newsockfd = accept(sockfd, (struct sockaddr *)&cli_addr,
    &clilen)) < 0) {
            perror("accept failed");
            continue;
        }
        // Legge i dati in modo bloccante
        while (1) {
            int n = read(newsockfd, buffer, BUFFER_SIZE - 1);
            if (n <= 0) {
                printf("Client disconnesso\n");
                break;
            }
            buffer[n] = '\0';
            printf("Ricevuto: %s", buffer);
            write(newsockfd, "OK, RICEVUTO\n", 4);
        }
        close(newsockfd);
```

Server Non-Bloccante (con Attesa Attiva)

```
1  while (1) {
2          // Accetta connessioni in modalita' non bloccante
3          newsockfd = accept(sockfd, (struct sockaddr *)&cli_addr, &
       clilen);
4          if (newsockfd < 0) {
5              if (errno == EWOULDBLOCK || errno == EAGAIN) {
6                  // Nessuna connessione in arrivo: continua il loop
7                  continue;
8              } else {
9                  perror("accept failed");
10                 exit(EXIT_FAILURE);
11             }
12         }
13
14         printf("Nuovo client connesso\n");
15         set_nonblocking(newsockfd); // Client socket non bloccante
```

Server Non-Bloccante (con Attesa Attiva)

```
1         // Legge i dati in un loop con attesa attiva
2         while (1) {
3             int n = recv(newsockfd, buffer, BUFFER_SIZE - 1, 0);
4             if (n > 0) {
5                 buffer[n] = '\0';
6                 printf("Ricevuto: %s", buffer);
7                 send(newsockfd, "ACK\n", 4, 0);
8             } else if (n == 0) {
9                 printf("Client disconnesso\n");
10                break;
11            } else {
12                if (errno == EWOULDBLOCK || errno == EAGAIN) {
13                    // Nessun dato disponibile: continua il loop
14                    continue;
15                } else {
16                    perror("recv failed");
17                    break;
18                }
19            }
```

Conclusioni

1. **Server sincrono (bloccante)**: Ideale per scenari semplici, dove la priorità è la semplicità del codice e non il carico elevato.
2. **Server con attesa attiva (non bloccante)**: Utile per applicazioni che richiedono massima reattività, ma inefficiente a causa del polling continuo.
3. **Asincronicità reale (oggetto della prossima lezione)**: La **soluzione ottimale** per scalabilità e efficienza è usare select()/epoll in combinazione con socket non bloccanti, evitando il busy-waiting.