

Reti informatiche cod. 545II [9 CFU]

Corso di Laurea in Ingegneria Informatica

Laboratorio e Programmazione di Rete
A. A. 2020/2021

Francesco Pistolesi, PhD
Dipartimento di Ingegneria dell'Informazione
francesco.pistolesi@unipi.it

Cosa impariamo oggi?

- Programmazione concorrente
- Socket non bloccanti
- I/O multiplexing

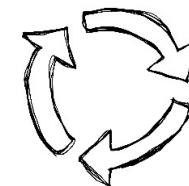
Server concorrenti



Tipi di (processi) server

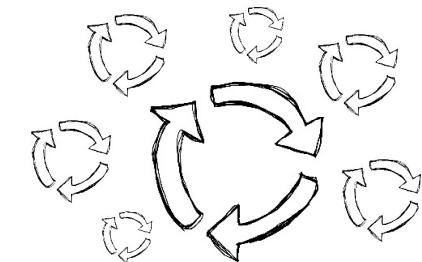
- **Server *iterativo***

- Serve **una richiesta** alla volta
- Per ogni richiesta accettata (`accept()`), il processo server la elabora e accoda le richieste che sopraggiungono



- **Server *concorrente***

- Serve **più richieste** alla volta
- Per ogni richiesta accettata (`accept()`) il processo server crea un processo che la elabora (detto *processo figlio*)



Primitiva `fork()`

- **Clona il processo**, il processo clone (*figlio*) esegue lo stesso codice del chiamante (*padre*)

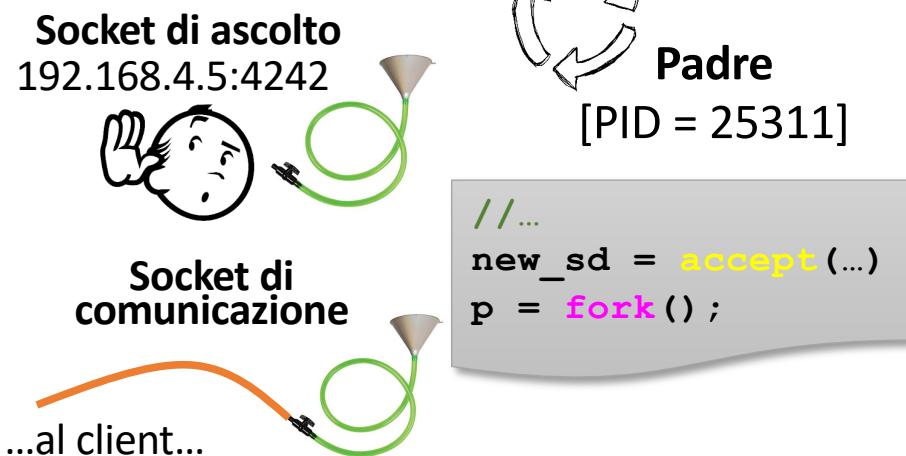
```
#include <unistd.h>
pid_t fork(void);
```

man 2 fork

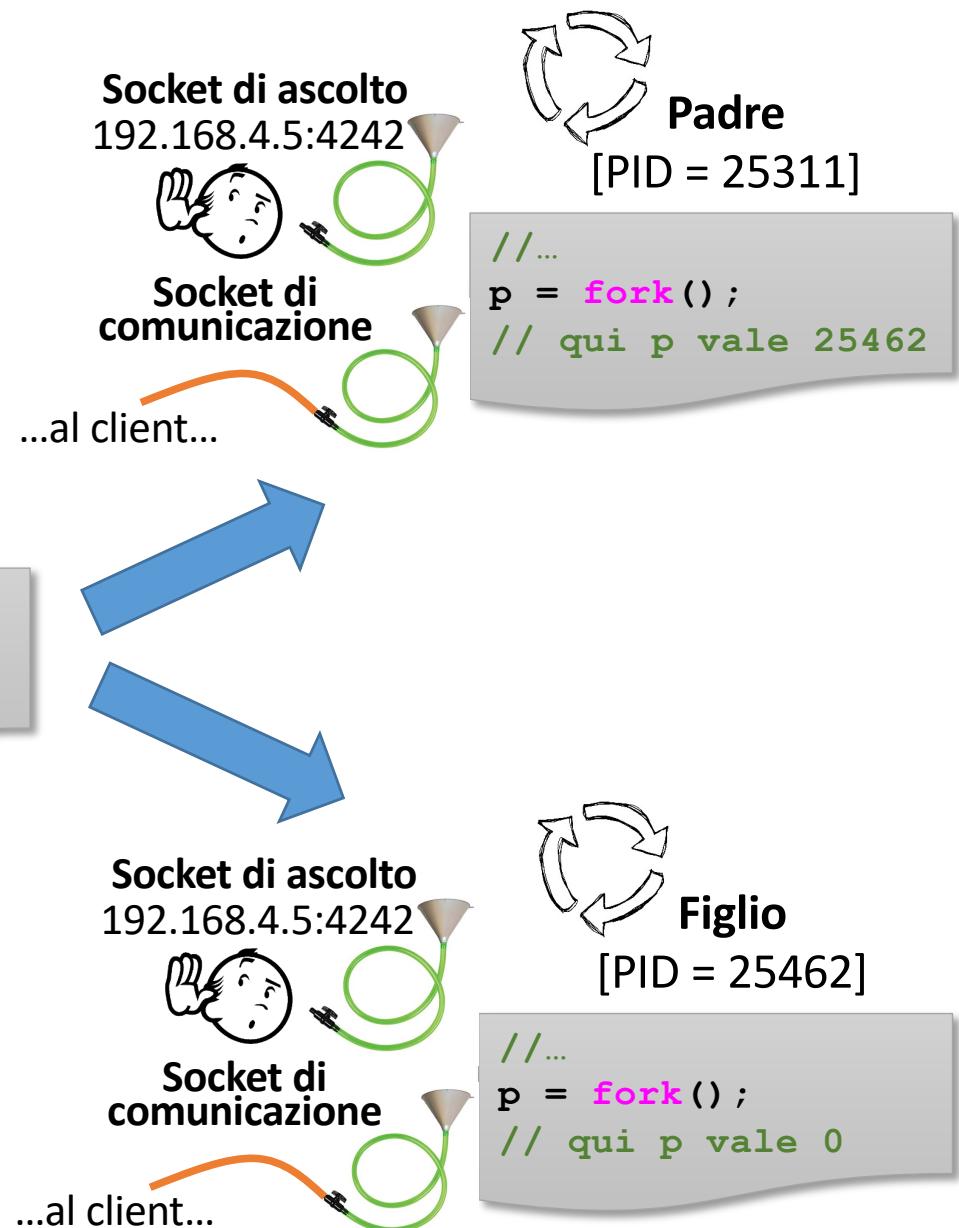
- Nel processo padre, `fork()` restituisce il *process identifier (PID)* del processo figlio creato
- Nel processo figlio, restituisce 0
- Restituisce -1 in caso di errore

Il tipo `pid_t` è un intero con segno. Nella GNU C Library è un `int`.

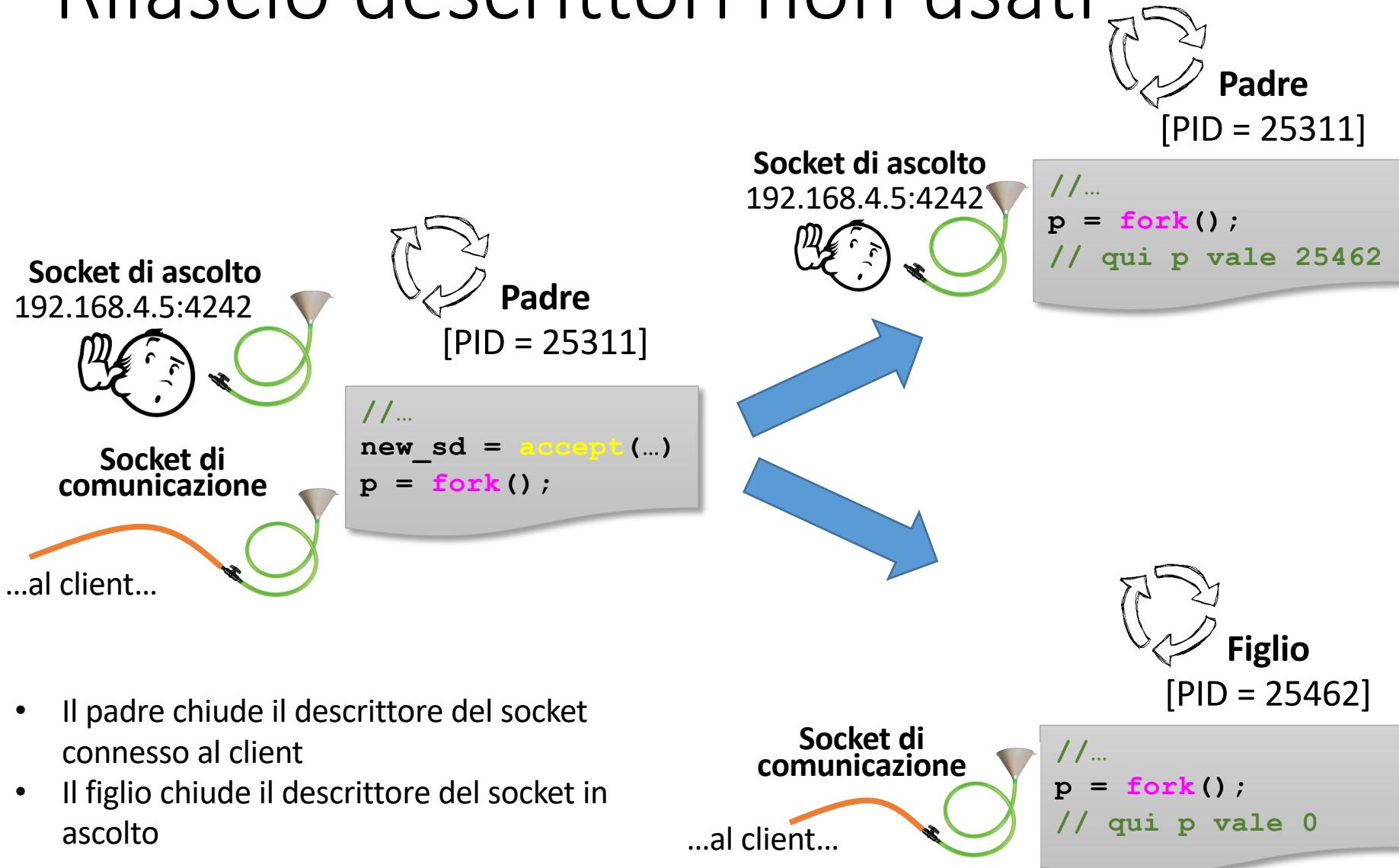
Fork di processo



Quando il processo è duplicato, padre e figlio si ritrovano gli stessi descrittori dei socket, duplicati



Rilascio descrittori non usati



Uso di `fork()`

```
pid_t pid;
//...
while(1) {
    new_sd = accept(sd, ...);
    pid = fork();

    // se pid vale 0 siamo nel FIGLIO
    if (pid == 0) {
        // chiusura del socket sd (il figlio non lo usa)      FIGLIO
        close(sd);

        // elaborazione della richiesta (usando il socket new_sd)

        // chiusura del socket new_sd
        close(new_sd);

        // il figlio termina
        exit(0);
    }
    // qui pid≠0, quindi siamo nel PADRE
    // chiusura del socket new_sd (il padre non lo usa)
    close(new_sd);

    // e fu sera e fu mattina...
}
```

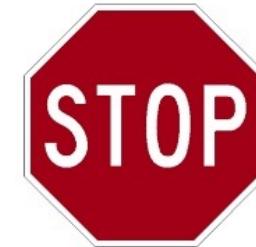
Serverino multi-processo

```
#include ...
int main () {
    int ret, sd, new_sd, len;
    struct sockaddr_in my_addr, cl_addr;
    //...
    pid_t pid;
    sd = socket(AF_INET, SOCK_STREAM, 0);
    /* Creazione indirizzo */
    ret = bind(sd, (struct sockaddr*)&my_addr, sizeof(my_addr));
    ret = listen(sd, 10);
    len = sizeof(cl_addr);
    while(1) {
        new_sd = accept(sd, (struct sockaddr*)&cl_addr, &len);
        pid = fork();
        if (pid == -1) {/* Gestione errore */}
        if (pid == 0) { /* Sono nel processo figlio */
            close(sd);
            /* Gestione richiesta (send, recv, ...) */
            close(new_sd);
            exit(0);
        }
        // Sono nel processo padre
        close(new_sd);
    }
}
```

Modelli di I/O

Socket bloccanti e non bloccanti

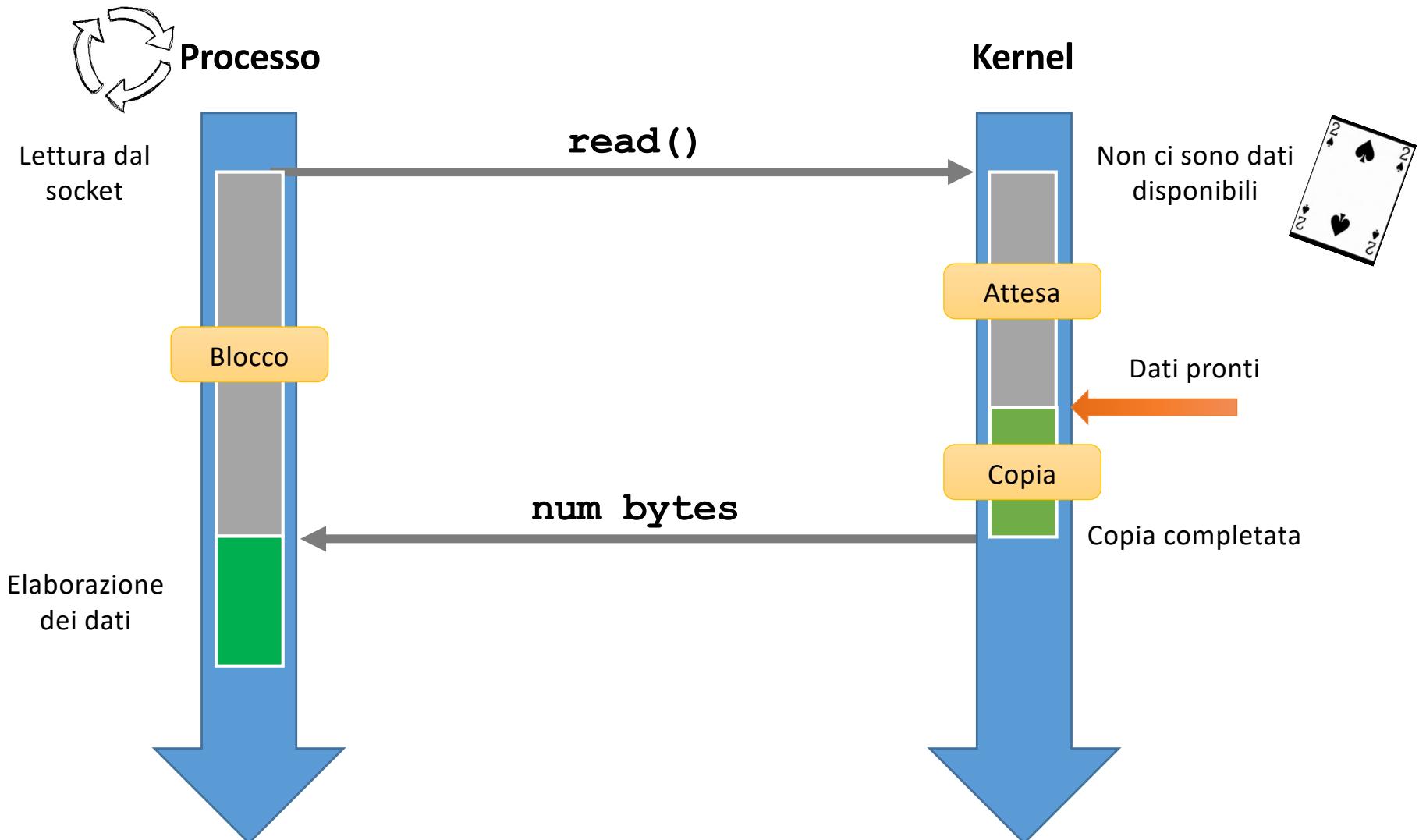
Socket bloccanti



- Di default, un socket è **bloccante**
 - `connect()` blocca il processo finché il socket non è connesso
 - `accept()` blocca il processo finché non arriva una richiesta di connessione
 - `send()` blocca il processo finché tutto il messaggio non è stato inviato (il buffer di invio potrebbe essere pieno)
 - `recv()` blocca il processo finché non ci sono dati disponibili o finché tutto il messaggio richiesto non è disponibile (flag `MSG_WAITALL`)

Socket bloccante

↓ timeline



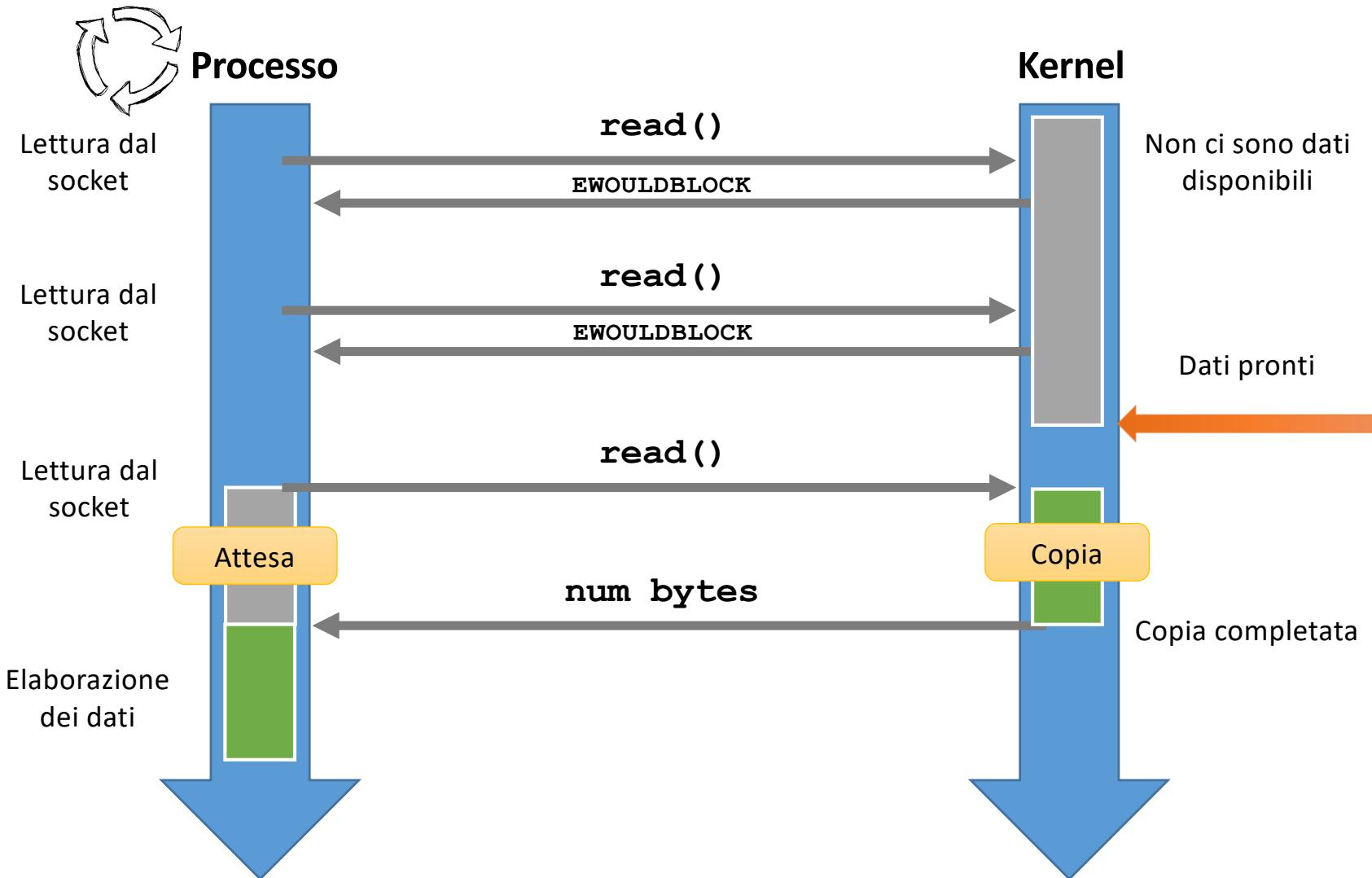
Socket non bloccante

- Un socket può essere settato come **non bloccante**

```
socket(AF_INET, SOCK_STREAM | SOCK_NONBLOCK, 0) ;
```

- **connect()**, se non può connettersi, restituisce -1 e imposta **errno** a **EINPROGRESS**
- **accept()**, se non ci sono richieste, restituisce -1 e imposta **errno** a **EWOULDBLOCK**
- **send()**, se non può inviare tutto il messaggio (il buffer è pieno), restituisce -1 e imposta **errno** a **EWOULDBLOCK**
- **recv()**, se non ci sono messaggi, restituisce -1 e imposta **errno** a **EWOULDBLOCK**

Socket non bloccante



I/O multiplexing

Gestire più descrittori/socket contemporaneamente

Multiplexing I/O sincrono

- Problema:
 - Se faccio operazioni su un socket bloccante, non posso controllarne altri



- Soluzione:
 - **Controllare più descrittori/socket** allo stesso tempo
 - **Multiplexing** con la primitiva **select()**: esamina più socket, il primo che è pronto viene usato

Descrittori pronti

- Un socket è pronto ***in lettura*** se:
 - C'è almeno un byte da leggere
 - Il socket è stato chiuso (`read()` restituirà 0)
 - È un socket in ascolto, e ci sono connessioni effettuate
 - C'è un errore (`read()` restituirà -1)
- Un socket è pronto ***in scrittura*** se:
 - C'è spazio nel buffer per scrivere
 - C'è un errore (`write()` restituirà -1)
 - Se il socket è chiuso, `errno` vale **EPIPE**

Insiemi di descrittori

- Un **descrittore** è un `int` da 0 (standard input) a `FD_SETSIZE` (di solito 1024)
- Un **insieme di descrittori** (detto `set`) si rappresenta con una variabile di tipo `fd_set`
 - Si manipola con delle *macro*:

```
/* Aggiungere un descrittore "fd" all'insieme "set" */
void FD_SET(int fd, fd_set* set);

/* Controllare se un descrittore "fd" è nell'insieme "set" */
int FD_ISSET(int fd, fd_set* set);

/* Rimuovere un descrittore "fd" dall'insieme "set" */
void FD_CLR(int fd, fd_set* set);

/* Svuotare l'insieme "set" */
void FD_ZERO(fd_set* set);
```

Primitiva `select()`

- Controlla **più socket**, rilevando quelli pronti

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(int nfds, fd_set* readfds, fd_set* writefds,
           fd_set* exceptfds, struct timeval* timeout);
```

man 2 select

- **nfds**: numero del descrittore più alto tra quelli da controllare, +1
- **readfds/writefds**: lista di descrittori da controllare per la lettura/scrittura
- **exceptfds**: lista di descrittori da controllare per le eccezioni (non ci interessa)
- **timeout**: intervallo di timeout
- Restituisce il **numero di descrittori pronti** (-1 in caso di errore)
- È **bloccante**: si blocca finché uno dei descrittori controllati non è pronto, o finché non scade il timeout (in questo caso, restituisce 0, cioè 0 descrittori pronti)

Struttura per il timeout

```
#include <sys/socket.h>
#include <netinet/in.h>

struct timeval {
    long tv_sec;      /* seconds */
    long tv_usec;     /* microseconds */
};
```

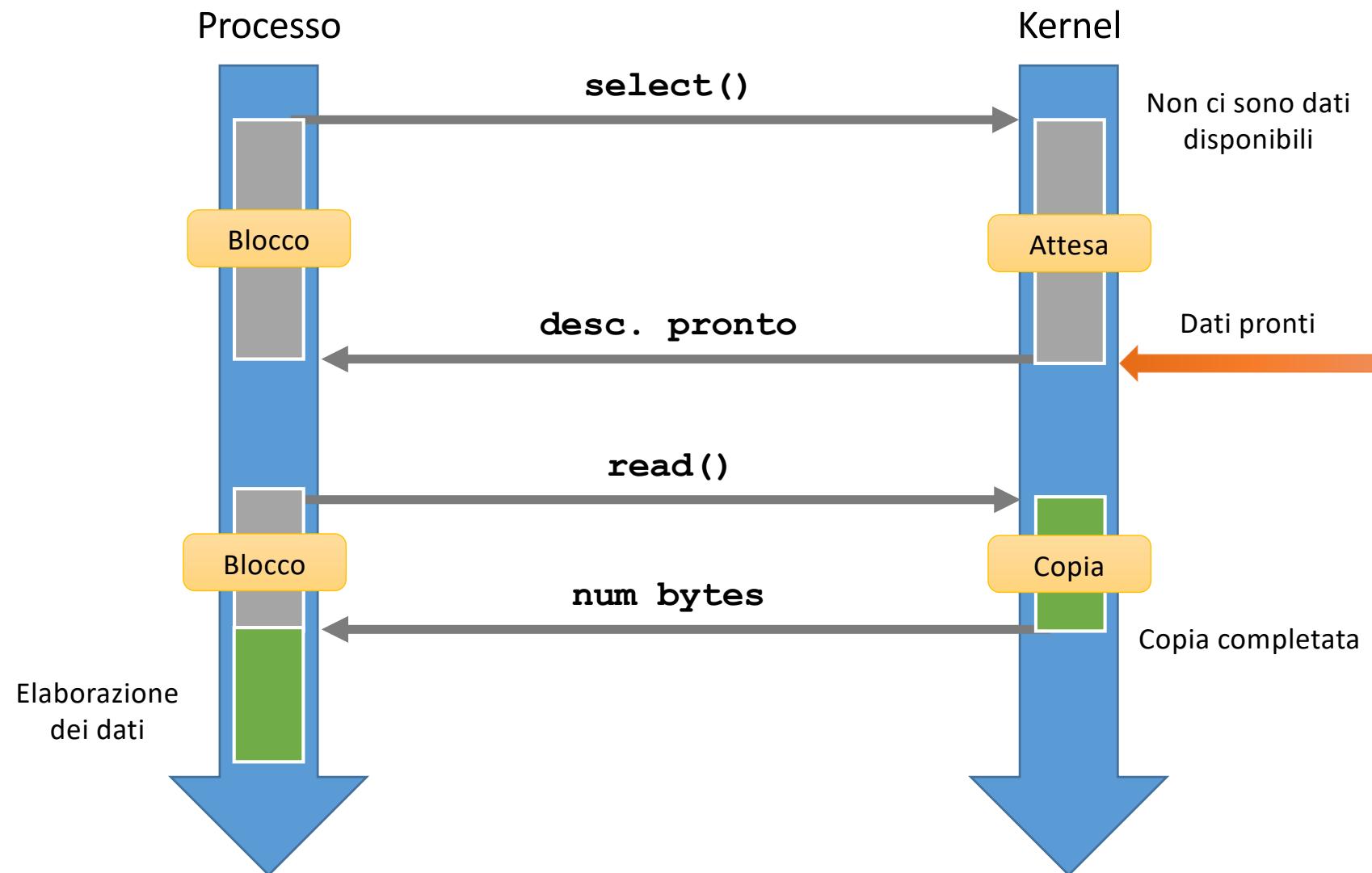
- **timeout = NULL**
 - Attesa infinita, fino a quando un descrittore è pronto
- **timeout = { 10; 5; }**
 - Attesa massima di 10 secondi e 5 microsecondi
- **timeout = { 0; 0; }**
 - Attesa nulla, controlla i descrittori ed esce immediatamente (*polling*)

Comportamento di `select()`

- `select()` modifica gli insiemi di descrittori:
 - **prima** di chiamare `select()`, occorre inserire i descrittori da monitorare nei set di lettura e di scrittura
 - **dopo** l'esecuzione di `select()`, i set di lettura e scrittura contengono i descrittori pronti



Multiplexing I/O sincrono



Utilizzo di `select()`

```
int main(int argc, char* argv[]){  
  
    fd_set master;                                /* Set principale gestito dal  
                                                    programmatore con le macro */  
  
    fd_set read_fds;                             /* Set di lettura gestito dalla  
                                                    select */  
  
    int fdmax;                                    // Numero max di descrittori  
  
    struct sockaddr_in sv_addr;      // Indirizzo server  
    struct sockaddr_in cl_addr;      // Indirizzo client  
    int listener;                            // Socket per l'ascolto  
    int newfd;                                // Socket di comunicazione  
    char buf[1024];                           // Buffer di applicazione  
    int nbytes;  
    int addrlen;  
    int i;  
/* Azzero i set */  
FD_ZERO(&master);  
FD_ZERO(&read_fds);  
  
    listener = socket(AF_INET, SOCK_STREAM, 0);
```

Utilizzo di select()

```
sv_addr.sin_family = AF_INET;
// INADDR_ANY mette il server in ascolto su tutte le
// interfacce (indirizzi IP) disponibili sul server
sv_addr.sin_addr.s_addr = INADDR_ANY;
sv_addr.sin_port = htons(20000);

bind(listener, (struct sockaddr*)& sv_addr, sizeof(sv_addr));
listen(listener, 10);
// Aggiungo il listener al set dei socket monitorati
FD_SET(listener, &master);
// Tengo traccia del maggiore (ora è il listener)
fdmax = listener;
for(;;){
    read_fds = master; // read_fds sarà modificato dalla select
    select(fdmax + 1, &read_fds, NULL, NULL, NULL);
    for(i=0; i<=fdmax; i++) { // f1) Scorro il set
        if(FD_ISSET(i, &read_fds)) { // i1) Trovato desc. pronto
            if(i == listener) { // i2) È il listener
                addrlen = sizeof(cl_addr);
                newfd = accept(listener,
                                (struct sockaddr *)&cl_addr, &addrlen)
                FD_SET(newfd, &master); // Aggiungo il nuovo socket
                if(newfd > fdmax){ fdmax = newfd; } // Aggiorno fdmax
            }
        }
    }
}
```

Utilizzo di `select()`

```
    }
    else { // Il socket connesso è pronto
        nbytes = recv(i, buf, sizeof(buf));
        //... Uso i dati
        // Chiudo il socket connesso
        close(i);
        // Tolgo il descrittore del socket connesso dal
        // set dei monitorati
        FD_CLR(i, &master);
    }
} // Fine if i1
} // Fine for f1
} // Fine for(;;)
return 0;
}
```

Esercizi

Utilità: leggere un'intera riga

fgets() legge dal file il cui puntatore è *fp* (restituito da *fopen*) *length*-1 caratteri (perché la stringa termina con ‘/0’) li copia in *stringa*. Restituisce un puntatore alla stringa; NULL in caso di errore.

```
char* fgets(char* stringa, int length, FILE* fp)
```

La variabile del C corrispondente allo **standard input** è FILE* stdin, definita in *<stdio.h>* (Il descrittore dello standard input è lo 0).

Esempio di lettura da tastiera:

```
#define BUFFER_SIZE 1024
...
int main(int argc, char* argv[]){
    char buffer[BUFFER_SIZE];
    ...
    fgets(buffer, BUFFER_SIZE, stdin);
}
```

Esercizio 1: Echo Server Concorrente

- a) Rendere multi-processo il server dell'esercizio 2 della volta precedente, con la primitiva **fork()**
- b) Rimuovere il limite dei 20 byte:
 1. il client invia la dimensione esatta della stringa
 2. il server legge la dimensione esatta di byte.
 3. Come fa il server a sapere in anticipo quanti byte leggere? Vedere slide successiva...

Inviare/ricevere la dimensione

```
uint16_t lmsg;  
...  
// Determino la dimensione dei dati che invierò  
len = strlen(buffer);  
lmsg = htons(len);  
// Invio la dimensione dei dati che invierò  
ret = send(sd, (void*) &lmsg, sizeof(uint16_t), 0);  
// Invio i dati  
ret = send(sd, (void*) buffer, len, 0);
```

Client

```
...  
// Ricevo la quantità di dati  
ret = recv(new_sd, (void*)&lmsg, sizeof(uint16_t), 0);  
// Rinconverto la dimensione in formato host  
len = ntohs(lmsg);  
// Ricevo i dati  
ret = recv(new_sd, (void*)buffer, len, 0);
```

Server

Esercizio 2: Time Server TCP

- Implementare un server TCP che periodicamente invia l'ora ai client che *si registrano* al servizio
 - Il server, periodicamente, controlla se c'è una richiesta di registrazione da parte di un client; se c'è, registra il client
 - I client rimangono connessi e periodicamente inviano una richiesta per ricevere l'ora e aspettano la risposta, stampandola a video ogni volta che arriva
 - L'ora è inviata in formato hh:mm:ss
- Usare la primitiva `select()` per gestire le richieste da parte dei client