```c
#include <stdio.h>, <stdlib.h>, <unistd.h>, <errno.h>, <sys/wait.h>, <pthread.h>, <string.h>, <time.h>, <semaphore.h>
#include <fcntl.h>, <sys/mman.h>, <sys/types.h>, <arpa/inet.h>, <netinet/in.h>, <sys/socket.h>, "common.h"
//fork processi
for (i = 0; i < n; i++) {
    pid_t pid = fork();
    if (pid == -1) {
        fprintf(stderr, "Can't fork, error %d\n", errno);
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        do_work();
        _exit(EXIT_SUCCESS);
    } else {
        wait(0);
    }
}

//creazione threads che eseguono thread_fun + join
for (i = 0; i < n; i++) {
    pthread_t thread;
    ret = pthread_create(&thread, NULL, thread_fun, NULL);
    if (ret != 0) {
        fprintf(stderr, "Can't create a new thread, error %d\n", ret);
        exit(EXIT_FAILURE);
    }
    ret = pthread_join(thread, NULL);
    if (ret != 0) {
        fprintf(stderr, "Cannot join on thread, error %d\n", ret);
        exit(EXIT_FAILURE);
    }
}

//threads con passaggio parametri e thread_id
shared_array = (unsigned long int*)calloc(n, sizeof(unsigned long int));
pthread_t* threads = (pthread_t*)malloc(n * sizeof(pthread_t));
int* thread_ids = (int*)malloc(n * sizeof(int));
int i;
for (i = 0; i < n; i++) {
    thread_ids[i] = i;
    if (pthread_create(&threads[i], NULL, thread_work, &thread_ids[i]) != 0) {
        fprintf(stderr, "Can't create a new thread, error %d\n", errno);
        exit(EXIT_FAILURE);
    }
}
unsigned long int computed_value = 0;
for (i = 0; i < n; i++) {
    pthread_join(threads[i], NULL);
    computed_value += shared_array[i];
}
free(shared_array);
free(threads);
free(thread_ids);

void* thread_work(void *arg) {
    int thread_idx = *((int*)arg);
    int i;
    for (i = 0; i < m; i++)
        shared_array[thread_idx] += v;
    return NULL;
}
```

```c
//thread con valore di ritorno
void* thread_work(void *arg) {
    unsigned long * portion = (unsigned long *) calloc(1,sizeof(unsigned long));
    for (int i = 0; i < m; i++)
        *portion += v;
    pthread_exit(portion);
}

unsigned long computed_value = 0, *thread_result;
for (i = 0; i < n; i++) {
    pthread_join(threads[i], (void**) &thread_result);
    computed_value += *thread_result;
    free(thread_result);
}

//semafori
typedef struct thread_args_s { //struttura per passaggio parametri
    int ID; sem_t* semaphore; int num_tasks;
} thread_args_t;

void* client(void* arg_ptr) {
    thread_args_t* args = (thread_args_t*) arg_ptr;
    int ret = sem_wait(args->semaphore);
    if (ret) {
        fprintf(stderr,"[FATAL ERROR] Could not lock the semaphore from thread %d: %s\n", args->ID, strerror(errno));
        exit(1);
    }
    //SEZIONE CRITICA
    ret = sem_post(args->semaphore);
    if (ret) {
        fprintf(stderr,"[FATAL ERROR] Could not unlock semaphore from thread %d: %s\n", args->ID, strerror(errno));
        exit(1);
    }
    free(args); return NULL;
}

int main(int argc, char* argv[]) {
    sem_t* semaphore = malloc(sizeof(sem_t));
    int ret = sem_init(semaphore, 0, NUM_RESOURCES);
    if (ret) {
        fprintf(stderr,"[FATAL ERROR] Could not create a semaphore: %s\n", strerror(errno));
        exit(1);
    }
    for (i = 0; i < THREAD_BURST; ++i) {
        pthread_t thread_handle;
        thread_args_t* args = malloc(sizeof(thread_args_t));
        args->semaphore = semaphore;
        args->ID = thread_ID;
        args->num_tasks = NUM_TASKS;
        if (pthread_create(&thread_handle, NULL, client, args)) {
            fprintf(stderr,"==> [DRIVER] ERROR: cannot create thread: %s\nExiting...\n", strerror(errno));
            exit(1);
        }
        ++thread_ID;
        pthread_detach(thread_handle); /*Non aspetterò questo thread. Se invece aspettassi tutti i threads con le
                              join sarei sicuro di non eliminare un semaforo prima che ogni thread lo abbia usato*/
    }
    sem_destroy(semaphore); free(semaphore);
}
```

```c
//producer-consumer con thread e semafori
sem_t empty_sem, fill_sem;
sem_t read_sem;
sem_t write_sem;
int transactions[BUFFER_SIZE];
int deposit = INITIAL_DEPOSIT;
int read_index, write_index;

void* performTransactions(void* x) { // producer thread
    if (sem_wait(&empty_sem)) {handle_error("Producer: sem_wait error empty sem");} //make sure there is space
    if (sem_wait(&write_sem)) {handle_error("Producer: sem_wait error write sem");} //get exclusive write access
    transactions[write_index] = performRandomTransaction();
    write_index = (write_index + 1) % BUFFER_SIZE;
    if (sem_post(&write_sem)) {handle_error("Producer: sem_post error write sem");} //release exclusive write access
    if (sem_post(&fill_sem)) {handle_error("Producer: sem_post error fill sem");} //notify that new element available
    pthread_exit(NULL);
}

void* processTransactions(void* x) { //consumer thread
    if (sem_wait(&fill_sem)) { // make sure there is data to consume
        handle_error("Consumer: sem_wait error fill sem");}
    if (sem_wait(&read_sem)) { // get exclusive read access
        handle_error("Consumer: sem_wait error read sem");}
    deposit += transactions[read_index];
    read_index = (read_index + 1) % BUFFER_SIZE;
    if (sem_post(&read_sem)) { // release exclusive read access
        handle_error("Consumer: sem_post error read sem");}
    if (sem_post(&empty_sem)) { // notify that a free cell in the buffer just became available
        handle_error("Consumer: sem_post error empty sem");}
    pthread_exit(NULL);
}

int main(int argc, char* argv[]) {
    read_index  = 0;
    write_index = 0;
    if (sem_init(&fill_sem, 0, 0)) handle_error("init error fill sem");
    if (sem_init(&empty_sem, 0, BUFFER_SIZE))  handle_error("init error empty sem");
    if (sem_init(&read_sem, 0, 1))  handle_error("init error read sem");
    if (sem_init(&write_sem, 0, 1))  handle_error("init error write sem");
    int ret;
    pthread_t producer[NUM_PRODUCERS], consumer[NUM_CONSUMERS];
    for (int i=0; i<NUM_PRODUCERS; ++i) {
        ret = pthread_create(&producer[i], NULL, performTransactions, arg);
        if (ret != 0)  handle_error_en(ret,"Error in pthread create (producer)");}
    for (int j=0; j<NUM_CONSUMERS; ++j) {
        ret = pthread_create(&consumer[j], NULL, processTransactions, arg);
        if (ret != 0) handle_error_en(ret,"Error in pthread create (consumer)");}
    for (i=0; i<NUM_PRODUCERS; ++i) {
        ret = pthread_join(producer[i], NULL);
        if (ret != 0) handle_error_en(ret,"Error in pthread join (producer)");}
    for (j=0; j<NUM_CONSUMERS; ++j) {
        ret = pthread_join(consumer[j], NULL);
        if (ret != 0) handle_error_en(ret,"Error in pthread join (consumer)");}
    if (sem_destroy(&fill_sem)) handle_error("Fill sem destroy error");
    if (sem_destroy(&empty_sem)) handle_error("Empty sem destroy error");
    if (sem_destroy(&read_sem)) handle_error("read sem destroy error");
    if (sem_destroy(&write_sem)) handle_error("write sem destroy error");
    exit(EXIT_SUCCESS);
}
```

```c
//server crea named semaphore e client lo usa con thread diversi
#define SEMAPHORE_NAME "/simple_scheduler" //in client.c

void* client(void *arg_ptr) { //in client.c
    sem_t* my_named_semaphore = sem_open(SEMAPHORE_NAME, 0);
    if (my_named_semaphore == SEM_FAILED) handle_error("Could not open the named semaphore from thread");
    if (sem_wait(my_named_semaphore)) handle_error("Could not lock the semaphore from thread");
    //SEZIONE CRITICA
    if (sem_post(my_named_semaphore)) handle_error("Could not unlock the semaphore from thread");
    if(sem_close(my_named_semaphore)) handle_error("Could not close the semaphore from thread");
    return NULL;
}

int main(int argc, char* argv[]) { //in client.c
    int thread_ID = 0;
    for (int i = 0; i < THREAD_BURST; ++i) {
        pthread_t thread_handle;
        int ret = pthread_create(&thread_handle, NULL, client, NULL);
        if (ret) handle_error_en(ret, "Cannot create thread");
        ++thread_ID;
        ret = pthread_detach(thread_handle);
        if (ret) {
            handle_error_en(ret, "Cannot detach thread");
            exit(1);
        }
    }
    pthread_exit(NULL);
}

#define SEMAPHORE_NAME "/simple_scheduler" //in server.c
sem_t* named_semaphore;

void cleanup() { //in server.c
    if (sem_close(named_semaphore)) handle_error("sem_close error");
    if (sem_unlink(SEMAPHORE_NAME)) handle_error("sem_unlink error");
    exit(0);
}

int main(int argc, char* argv[]) { //in server.c
    int ret;
    sem_unlink(SEMAPHORE_NAME);
    named_semaphore = sem_open(SEMAPHORE_NAME, O_CREAT | O_EXCL, 0600, NUM_RESOURCES);
    if (named_semaphore == SEM_FAILED) handle_error("Could not open the named semaphore");
    setQuitHandler(&cleanup); //cattura CTRL+C e manda a cleanup
    int current_value;
    ret = sem_getvalue(named_semaphore, &current_value);
    if (ret) {
        handle_error("Could not access the named semaphore");
        exit(1);
    }
    return 0;
}
```

```c
//scrittua in shared memory da due processi diversi
int * data; //naturalmente può anche essere una struct con buffer e read_index/wite_index
int fd;
#define SHM_NAME "/shmem-example"
#define SIZE (3 * sizeof(int))

int request() { //se partisse da un processo diverso non creato da fork dovrei fare anche la open
    if ((data = (int *) mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0))== MAP_FAILED)
        handle_error ("main: mmap");
    for (int i = 0; i < NUM; ++i) {
        data[i] = i;
    }
    if (munmap(data, SIZE) == -1)
        handle_error("main: munmap");
    return EXIT_SUCCESS;
}

int main(int argc, char **argv){
    shm_unlink(SHM_NAME);
    fd = shm_open(SHM_NAME, O_CREAT | O_EXCL | O_RDWR, 0600);
    if (fd < 0)
        handle_error("main: error in shm_open");
    if(ftruncate(fd, SIZE) == -1)
        handle_error ("main: ftruncate");
    pid_t pid = fork();
    if (pid == -1) {
        handle_error("main: fork");
    } else if (pid == 0) {
        request();
        _exit(EXIT_SUCCESS);
    }
    request();
    int ret = close(fd);
    if (ret == -1)
        handle_error("main: cannot close the shared memory");
    shm_unlink(SHM_NAME);
    _exit(EXIT_SUCCESS);
}

//copia da file in file con file descriptor
#define DEFAULT_BLOCK_SIZE  128;

static inline void performCopyBetweenDescriptors(int src_fd, int dest_fd, int block_size) {
    char* buf = malloc(block_size);
    while (1) {
        int read_bytes = 0;
        int bytes_left = block_size;
        while (bytes_left > 0) { //ciclo per leggere considerando EINTR
            int ret = read(src_fd, buf + read_bytes, bytes_left);
            if (ret == 0) break;
            if (ret == -1){
                if(errno == EINTR) // read() interrupted by a signal
                    continue;
                handle_error("Cannot read from source file");
            }
            bytes_left -= ret;
            read_bytes += ret;
        }
        if (read_bytes == 0) break;
        int written_bytes = 0;
        bytes_left = read_bytes;
        while (bytes_left > 0) {
            int ret = write(dest_fd, buf + written_bytes, bytes_left);
            if (ret == -1){
                if(errno == EINTR) // write() interrupted by a signal
                    continue;
                handle_error("Cannot write to destination file");
            }
            bytes_left -= ret;
            written_bytes += ret;
        }
    }
    free(buf);
}

int main(int argc, char* argv[]) {
    int block_size, src_fd, dest_fd;
    block_size = DEFAULT_BLOCK_SIZE;
    src_fd = open(argv[1], O_RDONLY);
    if (src_fd<0) handle_error("Could not open source file");
    dest_fd = open(argv[2], O_WRONLY | O_CREAT | O_EXCL, 0644);
    if (dest_fd < 0){
        if(errno == EEXIST) {
            fprintf(stderr, "WARNING: file %s already exists, I will overwrite it!\n", argv[2]);
            dest_fd = open(argv[2], O_WRONLY | O_CREAT, 0644);
        }else
            handle_error("Could not create destination file");
    }
    performCopyBetweenDescriptors(src_fd, dest_fd, block_size);
    int ret = close(src_fd);
    if (ret<0) handle_error("Could not close source file");
    ret = close(dest_fd);
    if (ret<0) handle_error("Could not close destination file");
    exit(EXIT_SUCCESS);
}


//pipe
int pipefd[2];

int write_to_pipe(int fd, const void *data, size_t data_len) {
    int written_bytes = 0, ret;
    while (written_bytes < data_len) {
        ret = write(fd, data + written_bytes, data_len - written_bytes);
        if (ret == -1 && errno == EINTR) continue;
        if (ret == -1) handle_error("error writing to pipe");
        written_bytes += ret;
    }
    return written_bytes;
}

int read_from_pipe(int fd, void *data, size_t data_len) {
    int read_bytes = 0, ret;
    while (read_bytes < data_len) {
        ret = read(fd, data + read_bytes, data_len - read_bytes);
        if (ret == -1 && errno == EINTR) continue;
        if (ret == -1) handle_error("error reading from pipe");
        if (ret ==  0) handle_error("unexpected close of the pipe");
        read_bytes += ret;
    }
    return read_bytes;
}

void reader() {
    int data[LUNGDATA];
    int ret = close(pipefd[1]);
    if(ret) handle_error("error closing pipe");
    for (int i = 0; i < NMESSAGGI; i++) {
        read_from_pipe(pipefd[0], data, sizeof(data));
    }
    ret = close(pipefd[0]);
    if(ret) handle_error("READER: error closing pipe");
}

void writer(int writer_id, sem_t* write_mutex) {
    int data[LUNGDATA];
    int ret = close(pipefd[0]);
    if(ret) handle_error("error closing pipe");
    for (int i = 0 ; i < NMESSAGGI; i++) {
        create_msg(data, MSG_ELEMS, i);
        write_to_pipe(pipefd[1], data, sizeof(data)); // sizeof(data) == MSG_ELEMS * sizeof(int)
    }
    ret = close(pipefd[1]);
    if(ret) handle_error("error closing pipe");
}

int main(int argc, char* argv[]) {
    int ret,i;
    pid_t pid;
    ret = pipe(pipefd);
    if(ret) handle_error("error creating the pipe");
    pid = fork();
    if (pid == -1) handle_error("error creating reader");
    if (pid == 0) {
        reader();
        _exit(0);
    }
    pid = fork();
    if (pid == -1) handle_error("error creating writer");
    if (pid == 0) {
        writer();
        _exit(0);
    }
    ret = close(pipefd[0]);
    if(ret) handle_error("error closing pipe");
    ret = close(pipefd[1]);
    if(ret) handle_error("error closing pipe");
    for (i = 0 ; i < 2; i++) { // shutdown phase
        int status;
        ret = wait(&status);
        if(ret == -1) handle_error("error waiting for a child to terminate");
        if (WEXITSTATUS(status)) {
            handle_error("ERROR: child process died unexpectedly");
        }
    }
    return 0;
}
```

```c
//named pipe ovvero FIFO
int readOneByOne(int fd, char* buf, char separator) { //legge fino al separator (\n)
    int ret;
    int bytes_read = 0;
    do {
        ret = read(fd, buf + bytes_read, 1);
        if (ret == -1 && errno == EINTR) continue;
        if (ret == -1) handle_error("Cannot read from FIFO");
        if (ret ==  0){
            printf("%s\n",buf);
            fflush(stdout);
            handle_error_en(bytes_read,"Process has closed the FIFO unexpectedly! Exiting...");
        }
    } while(buf[bytes_read++] != separator);
    return bytes_read;
}

void writeMsg(int fd, char* buf, int size) { //scrive
    int ret;
    int bytes_sent = 0;
    while (bytes_sent < size) {
        ret = write(fd, buf + bytes_sent, size - bytes_sent);
        if (ret == -1 && errno == EINTR) continue;
        if (ret == -1) handle_error("Cannot write to FIFO");
        bytes_sent += ret;
    }
}

static void cleanFIFOs(int echo_fifo, int client_fifo) {
    int ret = close(echo_fifo); // close the descriptor
    if(ret) handle_error("Cannot close Echo FIFO");
    ret = close(client_fifo); // close the descriptor
    if(ret) handle_error("Cannot close Client FIFO");
    ret = unlink(ECHO_FIFO_NAME); // destroy the FIFO
    if(ret) handle_error("Cannot unlink Echo FIFO");
    ret = unlink(CLNT_FIFO_NAME); // destroy the FIFO
    if(ret) handle_error("Cannot unlink Client FIFO");
}

int main(int argc, char* argv[]) { //in echo.c
    int ret, echo_fifo, client_fifo;
    char buf[1024];
    // Create the two FIFOs
    unlink(ECHO_FIFO_NAME);
    unlink(CLNT_FIFO_NAME);
    ret = mkfifo(ECHO_FIFO_NAME, 0666);
    if(ret) handle_error("Cannot create Echo FIFO");
    ret = mkfifo(CLNT_FIFO_NAME, 0666);
    if(ret) handle_error("Cannot create Client FIFO");
    echo_fifo = open(ECHO_FIFO_NAME, O_WRONLY); // Open the FIFO
    if(echo_fifo == -1) handle_error("Cannot open Echo FIFO for writing");
    client_fifo = open(CLNT_FIFO_NAME, O_RDONLY); // Open the FIFO
    if(client_fifo==-1) handle_error("Cannot open Client FIFO for reading");
    memset(buf,0,1024);
    int bytes_read = readOneByOne(client_fifo, buf, '\n');
    writeMsg(echo_fifo, buf, bytes_read);
    cleanFIFOs(echo_fifo, client_fifo);
    exit(EXIT_SUCCESS);
}
```

```c
int main(int argc, char* argv[]) { //in client.c
    int ret;
    int echo_fifo, client_fifo;
    char buf[1024];
    // Open the two FIFOs
    echo_fifo = open(ECHO_FIFO_NAME, O_RDONLY);
    if(echo_fifo == -1) handle_error("Cannot open Echo FIFO for reading");
    client_fifo = open(CLNT_FIFO_NAME, O_WRONLY);
    if(client_fifo == -1) handle_error("Cannot open Client FIFO for writing");
    memset(buf,0,1024);
    int msg_len = strlen(buf);
    writeMsg(client_fifo, buf, msg_len);
    bytes_read = readOneByOne(echo_fifo, buf, '\n');
    buf[bytes_read] = '\0';
    printf("Server response: %s", buf);
    // close the descriptors
    ret = close(echo_fifo);
    if(ret) handle_error("Cannot close Echo FIFO");
    ret = close(client_fifo);
    if(ret) handle_error("Cannot close Client FIFO");
    exit(EXIT_SUCCESS);
}

//socket
#define SERVER_ADDRESS  "127.0.0.1"
#define SERVER_COMMAND  "TIME"
#define SERVER_PORT     2015

void connection_handler(int socket_desc) { //in server.c
    int ret;
    char* allowed_command = SERVER_COMMAND;
    size_t allowed_command_len = strlen(allowed_command);
    char send_buf[256];
    // receive command from client
    char recv_buf[256];
    size_t recv_buf_len = sizeof(recv_buf);
    int recv_bytes;
    //non consideriamo messaggi ricevuti parzialmente
    while ( (recv_bytes = recv(socket_desc, recv_buf, recv_buf_len, 0)) < 0 ) {
        if (errno == EINTR) continue;
        handle_error("Cannot read from socket");
    }
    // parse command received and write reply in send_buf
    if (recv_bytes == allowed_command_len && !memcmp(recv_buf, allowed_command, allowed_command_len)) {
        time_t curr_time;
        time(&curr_time);
        sprintf(send_buf, "%s", ctime(&curr_time));
    } else {
        sprintf(send_buf, "INVALID REQUEST");
    }
    size_t server_message_len = strlen(send_buf);
    while ( (ret = send(socket_desc, send_buf, server_message_len, 0)) < 0 ) {
        if (errno == EINTR) continue;
        handle_error("Cannot write to the socket");
    }
    // close socket
    ret = close(socket_desc);
    if (ret<0) handle_error("Cannot close socket for incoming connection");
}
```

```c
int main(int argc, char* argv[]) { //in server.c
    int ret, socket_desc, client_desc;
    struct sockaddr_in server_addr = {0}, client_addr = {0};
    int sockaddr_len = sizeof(struct sockaddr_in);
    // initialize socket for listening
    socket_desc = socket(AF_INET , SOCK_STREAM , 0);
    if (socket_desc<0) handle_error("Could not create socket");
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_family      = AF_INET;
    server_addr.sin_port        = htons(SERVER_PORT);
    int reuseaddr_opt = 1;
    ret = setsockopt(socket_desc, SOL_SOCKET, SO_REUSEADDR, &reuseaddr_opt, sizeof(reuseaddr_opt));
    if (ret < 0) handle_error("Cannot set SO_REUSEADDR option");
    // bind address to socket
    ret = bind(socket_desc, (struct sockaddr*) &server_addr, sockaddr_len);
    if (ret < 0) handle_error("Cannot bind address to socket");
    // start listening
    ret = listen(socket_desc, MAX_CONN_QUEUE);
    if (ret < 0) handle_error("Cannot listen on socket");
    // loop to handle incoming connections serially
    while (1) {
        client_desc = accept(socket_desc, (struct sockaddr*) &client_addr, (socklen_t*) &sockaddr_len);
        if (client_desc < 0) handle_error("Cannot open socket for incoming connection");
        connection_handler(client_desc);
    }
    exit(EXIT_SUCCESS);
}

int main(int argc, char* argv[]) { //in client.c
    int ret, socket_desc;
    struct sockaddr_in server_addr = {0};
    socket_desc = socket(AF_INET, SOCK_STREAM, 0); // create a socket
    if (socket_desc < 0) handle_error("Could not create socket");
    // set up parameters for the connection
    server_addr.sin_addr.s_addr = inet_addr(SERVER_ADDRESS);
    server_addr.sin_family      = AF_INET;
    server_addr.sin_port        = htons(SERVER_PORT);
    // initiate a connection on the socket
    ret = connect(socket_desc, (struct sockaddr*) &server_addr, sizeof(struct sockaddr_in));
    if (ret < 0) handle_error("Could not create connection");
    // send command to server, non consideriamo messaggi inviati parzialmente
    char* command = SERVER_COMMAND;
    size_t command_len = strlen(command);
    while ((ret = send(socket_desc, command, command_len, 0)) < 0) {
        if (errno == EINTR) continue;
        handle_error("Cannot write to socket");
    }
    // read message from the server, non consideriamo messaggi ricevuti parzialmente
    char recv_buf[256];
    size_t recv_buf_len = sizeof(recv_buf);
    while ((ret = recv(socket_desc, recv_buf, recv_buf_len - 1, 0)) < 0 ) {
        if (errno == EINTR) continue;
        handle_error("Cannot read from socket");
    }
    // close the socket
    ret = close(socket_desc);
    if (ret < 0) handle_error("Cannot close socket");
    exit(EXIT_SUCCESS);
}
```

```c
//socket send e recv con controllo messaggi parzialmente ricevuti o inviati
if (fgets(buf, sizeof(buf), stdin) != (char*)buf) { //lettura da stdin
    fprintf(stderr, "Error while reading from stdin, exiting...\n"); exit(EXIT_FAILURE);
}
msg_len = strlen(buf);
bytes_sent=0;
while ( bytes_sent < msg_len) { //controlla messaggi parzialmente inviati
    ret = send(socket_desc, buf + bytes_sent, msg_len - bytes_sent, 0);
    if (ret == -1 && errno == EINTR) continue;
    if (ret == -1) handle_error("Cannot write to the socket");
    bytes_sent += ret;
}
//controlla se messaggio ricevuto corrisponde al comando di terminazione
if (msg_len == quit_command_len && !memcmp(buf, quit_command, quit_command_len)) break;
recv_bytes = 0;
do { //controlla messaggi parzialmente ricevuti e legge 1 byte alla volta fino a trovare '\n'
    ret = recv(socket_desc, buf + recv_bytes, 1, 0);
    if (ret == -1 && errno == EINTR) continue;
    if (ret == -1) handle_error("Cannot read from the socket");
    if (ret == 0) break;
} while ( buf[recv_bytes++] != '\n' );

//utilizzo socket UDP
bytes_sent=0;
while ( bytes_sent < msg_len) {
    ret = sendto(socket_desc, buf, msg_len, 0, (struct sockaddr*) &server_addr, sizeof(struct sockaddr_in));
    if (ret == -1 && errno == EINTR) continue;
    if (ret == -1) handle_error("Cannot write to the socket");
    bytes_sent = ret;
}
recv_bytes = 0;
do {
    ret = recvfrom(socket_desc, buf, buf_len, 0, NULL, NULL);
    if (ret == -1 && errno == EINTR) continue;
    if (ret == -1) handle_error("Cannot read from the socket");
    if (ret == 0) break;
    recv_bytes = ret;
} while ( recv_bytes<=0 );

//server multi-process, il main fa tutto e poi chiama la funzione passando il descrittore della socket
void mprocServer(int server_desc) {
    int ret = 0;
    struct sockaddr_in client_addr = {0};
    int sockaddr_len = sizeof(struct sockaddr_in);
    while (1) {
        int client_desc = accept(server_desc, (struct sockaddr *) &client_addr, (socklen_t *)&sockaddr_len);
        if (client_desc == -1 && errno == EINTR) continue;
        if (client_desc < 0) handle_error("Cannot open socket for incoming connection");
        pid_t pid = fork();
        if (pid < 0) handle_error("Cannot fork server process to handle the request");
        else if (pid == 0) {
            ret = close(server_desc);
            if (ret) handle_error("Cannot close listening socket in child process");
            connection_handler(client_desc, &client_addr);
            _exit(EXIT_SUCCESS);
        }
        else {ret = close(client_desc);
            if (ret) handle_error("Cannot close incoming socket in main process");
            memset(&client_addr, 0, sizeof(struct sockaddr_in));
}   }   }
```

```c
//server multi-thread, il main fa tutto e poi chiama la funzione passando il descrittore della socket
typedef struct handler_args_s {
    int socket_desc;
    struct sockaddr_in* client_addr;
} handler_args_t;

void *thread_connection_handler(void *arg) {
    handler_args_t *args = (handler_args_t *)arg;
    int socket_desc = args->socket_desc;
    struct sockaddr_in *client_addr = args->client_addr;
    connection_handler(socket_desc,client_addr);
    free(args->client_addr);
    free(args);
    pthread_exit(NULL);
}

void mthreadServer(int server_desc) {
    int ret = 0;
    int sockaddr_len = sizeof(struct sockaddr_in);
    while (1) {
        struct sockaddr_in* client_addr = calloc(1, sizeof(struct sockaddr_in));
        int client_desc = accept(server_desc, (struct sockaddr *) client_addr, (socklen_t *)&sockaddr_len);
        if (client_desc == -1 && errno == EINTR) continue;
        if (client_desc < 0) handle_error("Cannot open socket for incoming connection");
        pthread_t thread;
        handler_args_t *thread_args = malloc(sizeof(handler_args_t));
        thread_args->socket_desc = client_desc;
        thread_args->client_addr = client_addr;
        ret = pthread_create(&thread, NULL, thread_connection_handler, (void *)thread_args);
        if (ret) handle_error_en(ret, "Could not create a new thread");
        ret = pthread_detach(thread);
        if (ret) handle_error_en(ret, "Could not detach the thread");
    }
}
```

```c
//esercizio riepilogativo finale: semafori, multi-thread e socket

// array di N semafori per accedere in mutua esclusione ad N contatori
sem_t semaphores[N];
unsigned int shared_counters[N];

// struttura dati con gli argomenti per thread connection_handler
typedef struct handler_args_s {
    unsigned int client_id;
    int socket_desc;
    struct sockaddr_in* client_addr;
} handler_args_t;

unsigned int process_resource(unsigned int client_id, unsigned int resource_id) {
    int ret;

    ret = sem_wait(&semaphores[resource_id]);
    if(ret) handle_error("Errore nella wait");

    printf("Risorsa %u LOCKED dal client %u! Processamento in corso...\n", resource_id, client_id);

    /* inizio sezione critica */
    unsigned int counter_updated = ++shared_counters[resource_id];
    printf("Nuovo contatore per risorsa %u: %u\n", resource_id, counter_updated);
    sleep(SLEEP_TIME);
    /* fine sezione critica */

    ret = sem_post(&semaphores[resource_id]);

    printf("Risorsa %u UNLOCKED\n", resource_id);
    return counter_updated;
}

void* connection_handler(void* arg) {
    int ret;
    handler_args_t* args = (handler_args_t*)arg;
    int socket_desc = args->socket_desc;
    struct sockaddr_in* client_addr = args->client_addr;
    unsigned int client_id = args->client_id;
    char buf[1024];
    char* quit_command = SERVER_COMMAND;
    size_t quit_command_len = strlen(quit_command);

    // parso il client IP address e la porta
    char client_ip[INET_ADDRSTRLEN];
    inet_ntop(AF_INET, &(client_addr->sin_addr), client_ip, INET_ADDRSTRLEN);
    uint16_t client_port = ntohs(client_addr->sin_port); // port number is an unsigned short
    printf("Client %u connesso sulla porta %u \n", client_id, client_port);

    // ciclo di scambio messaggi
    unsigned int resource_id;
    while (1) {
        int bytes_read = 0;
        do {
            ret = recv(socket_desc, buf + bytes_read, 1, 0);
            if (ret == -1 && errno == EINTR) continue;
            if (ret == -1) handle_error("Errore nella lettura dalla socket");
            if (ret ==  0) break;
        }while(buf[bytes_read++] != '\n');
        size_t msg_len = bytes_read - 1; // NON MODIFICARE (vedi commenti sopra)

        // verifico se devo terminare (comando "QUIT" ricevuto, o endpoint chiuso)
        if ((msg_len == quit_command_len && !memcmp(buf, quit_command, quit_command_len)) || ret == 0) break;

        // avvio il processamento della risorsa, aggiorno il relativo contatore e lo salvo in buf
        buf[msg_len] = '\0';
        int tmp = atoi(buf); // devo usare un int per gestire numeri negativi
        resource_id = (tmp > 0 && tmp < N) ? tmp : 0;
        unsigned int counter_updated = process_resource(client_id, resource_id);

        sprintf(buf, "[risorsa %u] contatore: %u", resource_id, counter_updated);
        msg_len = strlen(buf);

        int bytes_sent = 0;
        while(bytes_sent < msg_len) {
            ret = send(socket_desc, buf+bytes_sent, msg_len-bytes_sent, 0);
            if (ret == -1 && errno == EINTR) continue;
            if (ret == -1) handle_error("Errore nella scrittura sulla socket");
            bytes_sent += ret;
        }
    }
    ret = close(socket_desc);
    if(ret) handle_error("Errore nella chiusura della socket");
    free(arg);

    printf("Thread connection_handler terminato\n");
    pthread_exit(NULL);
}

int main(int argc, char* argv[]) {
    int ret;
    int i;
    for(i = 0; i < N; i++) {
       ret = sem_init(&semaphores[i], 0, 1);
       if(ret) handle_error("Errore nell’inizializzazione del semaforo");
    }

    // creo le strutture per la socket
    int socket_desc, client_desc;
    struct sockaddr_in server_addr = {0};

    int sockaddr_len = sizeof(struct sockaddr_in); // da riusare per le accept()

    // inizializzo la socket
    socket_desc = socket(AF_INET , SOCK_STREAM , 0);
    if(socket_desc == -1) handle_error("Impossibile creare la socket");

    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_family      = AF_INET;
    server_addr.sin_port        = htons(SERVER_PORT);

    // abilito l'opzione SO_REUSEADDR per riavviare il server dopo un crash
    int reuseaddr_opt = 1;
    ret = setsockopt(socket_desc, SOL_SOCKET, SO_REUSEADDR, &reuseaddr_opt, sizeof(reuseaddr_opt));
    if(ret) handle_error("Impossibile settare l'opzione SO_REUSEADDR");
    // bind & listen
    ret = bind(socket_desc, (struct sockaddr*) &server_addr, sockaddr_len);
    if(ret) handle_error("Impossibile fare il binding indirizzo-socket");

    ret = listen(socket_desc, 16);
    if(ret) handle_error("Impossibile ascoltare dalla socket");

    // loop per gestire connessioni in ingresso con nuovi thread
    printf("Server pronto ad accettare connessioni!\n");
    unsigned int client_id = 0;

    while (1) {
        pthread_t thread;
        //alloco un buffer client_addr per la connessione
        struct sockaddr_in* client_addr = calloc(1, sizeof(struct sockaddr_in));

        // accetto connessioni in ingresso
        client_desc = accept(socket_desc, (struct sockaddr*) client_addr, (socklen_t*) &sockaddr_len);
        if (client_desc == -1 && errno == EINTR) continue; // check per segnali di interruzione
        if (client_desc == -1) handle_error("Impossibile aprire la socket per connessioni in ingresso.");
        printf("Connessione accettata\n");

        handler_args_t* arg = calloc(1, sizeof(handler_args_t));
        arg->socket_desc = client_desc;
        arg->client_addr = client_addr;
        arg->client_id = client_id;
        ret = pthread_create(&thread, NULL, connection_handler, arg);
        if(ret) handle_error_en(ret,"Errore nella creazione del thread");

        ret = pthread_detach(thread);
        if(ret) handle_error_en(ret,"Errore nel detach del thread");

        // incremento il client ID dopo ogni connessione accettata
        client_id++;
    }
    return 0;
}
```