# Week 5: Inter-process Communication

## Unix Sockets

Unix domain sockets represent a method of interprocess communication. Its use is quite similart to UNIX network sockets and they are using the same interface (API). They are also using functions that are shared with other methods - we will wxplain that later.

By using sockets, one of the processes acts like a server while the others operate as the clients. A server prepares an entry point (an address) and waits for clients to connect. Clients are then connected on a known address (if exists). When the connection is established, the sockets enable two-way communication. That means that both server and client can send or receive data.

**To provide an address we use the structure *sockaddr_un*.**

```
struct sockaddr_un{
    sa_family_t sun_family; // always AF_UNIX for a UNIX socket
    char sun_path[108]; // path to a socket (in file system or an abstract name)
}
```

Here, we have two members that we need to set. A value *sun_family* tells what kind of socket do we want to use. For UNIX sockets, we alway use a constant AF_UNIX.

In an array *sun_path* we put an address (C string, max. length of 107 characters). For UNIX sockets, this can be an absolute or a relative path to a file system, where the socket will be created. Otherwise, it can also be an abstract address which is started with '0'. In this case, socket will stay in a memory and will not be visible through the file system.

**To establish the connection and communication we use the following functions.**

```
int socket(int domain, int type, int protocol);
```

Function socket creates an unconnected socket. As a parameter we need to provide a family/domain of the socket (AF_UNIX for a UNIX socket), socket type/way of communication (SOCK_SEQPACKET for the separated messages) and protocol (NULL as default). It returns an integer which represents a handle to a newly created socket.

```
int bind(int socket, const struct sockaddr *address, socklen_t adress_len);
```

Function bind binds the created socket with an address. For UNIX sockets, if the address is free, it binds to it and creates a file (if the address is a path).

```
int listen(int socket, int backlog);
```

Function listen configures a socket for a listening. This function is used by a server in order to create a socket, where he then waits for the clients.

```
int accept(int socket, struct sockaddr *address, socklen_t* address_len);
```

Function accept waits for the upcoming client that wants to connect to a server. If successful, it returns a new handle for a communication with the connected client. Of course, this function is used by a server when he is ready for the next client.

```
int connect(int socket, struct sockaddr *address, socklen_t address_len);
```

Function connect is used by the client to connect on the server throught a handle. If server did not set an address for binding properly, the function will not be successful.

The following functions are used also with the other structures and methods, therefore they have a little bit different names of the parameters. Here, we use different namest just to make it more clear, which parameters should we provide in our context.

```
size_t read(int socket, void *data, size_t data_len);
```

Function read reads data from a socket. Data is read in written in a raw binary form (function is not aware if here there are strings, structures, and other stuff). Function returns a number of bytes read. Function is not finished until there are not enough data available. If the socket was already closed in other process, it returns 0.

```
size_t write(int socket, void *data, size_t data_len);
```

Function write writes data to a socket. Similarly as funciton read, also here we operate with raw binary data. Function returns a number of bytes written. If the socket was already closed in other process, writting will be unsuccessful.

```
int close(int socket);
```

Function close closes the socket that is open. A closure of a socket is visible also in other processes that use that socket (as stated above, writting and reading will be unsuccessful after closure).

## Attention!

When we prepare a way of communication between the processes, it is very important that we exactly determine the order of sending and receiving the messages (protocol). For example, if both server and client try to read a message simultaneously, they will both stop. Thus, if one process expects a message, the other process will have to send the message or close the socket and vice versa.

## An example

Here we have an example of a server and a client. We run the single instance of a server.

A server waits for clients on a relative path named "./communication_channel". For each of the client he waits for its message and sends * in a response. If a client sends a word STOP, the server finishes its work and is closed.

A client connects to a server and reads a shor message from a keyboard (standard input). It reads up to 10 characters which are then sent to a server. Then it reads and prints out the response from a server. It reads up to 10 characters that are sent to a server. If we write the word QUIT, a cliend closes its socket and quits.

We can run only one instance of a server. Any subsequent trials of running the server will be unsuccessful because the socket address (relative path) will be already taken.

However, clients can be run multiple times and even simultaneously. If the server is up and running and there exist an adress of the socket, the clients can connect to a server. The first one in line starts the communication with a server. Other clients then wait in queue to come into line.

```
/*
 * File server.c
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>
```

```
const char socket_name[]="./communication_channel";


int main(int argc, char *argv[])
{
    struct sockaddr_un addr;
    int stop_server = 0;
    int ret;
    int connection_socket;
    int data_socket;

    char msg[11];

    // Create a new socket
    printf("Creating a socket\n");
    connection_socket = socket(AF_UNIX, SOCK_SEQPACKET, 0);
    if (connection_socket == -1) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    // Address structure initialized to 0
    memset(&addr, 0, sizeof(struct sockaddr_un));

    // Set the family and name (path) of the socket (AF_UNIX)
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, socket_name, sizeof(addr.sun_path) - 1);

    // Bind the name and type with the socket, a file is created at this point
    // We create a socket for binding
    printf("Binding the address of the socket\n");
    ret = bind(connection_socket, (const struct sockaddr *) &addr,
            sizeof(struct sockaddr_un));
    if (ret == -1) {
        perror("bind");
        exit(-1);
    }

    // Configuration for waiting of the clients - up to 2 in the queue
    ret = listen(connection_socket, 0);
    if (ret == -1) {
        perror("listen");
        exit(-1);
    }

    // Loop for waiting on the clients to connect
    while(!stop_server){

        // We wait for a client, we obtain a new socket for a communication with it
        printf("Waiting for the client ...\n");
        data_socket = accept(connection_socket, NULL, NULL);
        if (data_socket == -1) {
            perror("accept");
            exit(-1);
        }

        // Loop for a communication with a single client
        while(1){
            // Wait for message
            printf("Reading the message ...\n");
            ret = read(data_socket, msg, 10);
            if (ret == -1) {
                perror("read");
                exit(-1);
            }
            if (ret == 0){
                break;
            }
            // Because we read a string of characters, we ensure that it should conclude with 0
            msg[ret]=0;
            printf("Message received: %s\n", msg);

            // If the message was STOP, the server should stop
            if (strcmp(msg, "STOP")==0) {
                stop_server=1;
                break;
            }
            // Response with *
            ret=write(data_socket, "*", 1);
            if (ret == -1) {
                perror("write");
                exit(-1);
            }
        }
        // Close communication socket
        close(data_socket);
    }
    // Close and erase the socket for establishing the connection
    close(connection_socket);
    unlink(socket_name);

    exit(-1);
}
```

```
/*
 * File client.c
 */

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

const char SOCKET_NAME[]="./communication_channel";

int main(int argc, char *argv[])
{
    struct sockaddr_un addr; // address
    int ret; // a returned value of several funtions
    int data_socket; // socket

    // Create a new socket
    printf("Creating a socket\n");
    data_socket = socket(AF_UNIX, SOCK_SEQPACKET, 0);
    if (data_socket == -1) {
        perror("Error by creating the socket\n");
        exit(-1);
    }
    // Addres structure initialized to 0
    memset(&addr, 0, sizeof(struct sockaddr_un));
    // A socket family is set - Unix socket for a local communication between the processes
    addr.sun_family = AF_UNIX;
    // name (path) of the socket copied into addr.sun_path
    strncpy(addr.sun_path, SOCKET_NAME, sizeof(addr.sun_path) - 1);

    // Establish the connection
    ret = connect (data_socket, (const struct sockaddr *) &addr, sizeof(struct sockaddr_un));
    if (ret == -1) {
        perror("Error by connecting to the server\n");
        exit(-1);
    }

    // If connection successful, we read a message from a standard input (keyboard)
    // Then the message is send to a server and after that we wait for a response from a server
    char msg[11];
    char answer[11];
    while(1){
        printf("Message? ");
        scanf("%10s", msg); // We read up to 10 characters

        // if we put in QUIT, the client is finished
        if(strcmp(msg, "QUIT")==0){
            break;
        }

        // Send a message to the server
        ret = write(data_socket, msg, strlen(msg));
        if (ret == -1) {
            perror("write");
            exit(-1);
        }
        // We recieve a message with max. 10 characters
        ret = read(data_socket, answer, 10);
        if (ret == -1) {
            perror("read");
            exit(-1);
        }

        // Function read works on binary data, not strings!
        // In order to do that, we must conclude it with 0
        answer[ret] = 0;
        printf("odgovor: %s\n", answer);
    }
    // Close the connection
    close(data_socket);
    return 0;
}
```

# Shared Memory

The most direct way to communicate between processes is by using shared memory. In this case, the virtual memory space section of the different processes is mapped to the same physical addresses. For these addresses, it is considered that the data in one process is visible to the other without or with minimal processing.

To use shared memory, we need some kind of access point for different processes and an appropriate configuration (i.e. allocation of memory). In POSIX compatible systems, POSIX shared memory is defined for the access point. With the appropriate configuration, we can use mmap function (which also has some other interesting functionalities).

## POSIX shared memory

POSIX shared memory has the same properties as anonymous shared memory. Its main advantage is that we do not need to store files on the disk for sharing memory - in some cases this is an undesirable feature. Since it does not write to the file, POSIX shared memory is also slightly faster.

We use the following functions to manage POSIX shared memory:

```
int shm_open(const char *name, int oflag, mode_t mode);
int shm_unlink(const char *name);
```

The function **shm_open** has the same functionality as the **open** function for opening files. It returns the file descriptor of a shared memory, which is then created by calling the **mmap** function. **Shm_unlink** is used to remove shared memory, so new processes can no longer access this shared memory. The **shm_unlink** call does not affect existing memory mappings - they remain active and usable until processed by **munmap**.

The example below prepares shared memory and then creates **L** children who counts the factorials. The results are stored in shared memory. The parent then displays the contents of the shared memory, i.e. results of calculations.

```
// gcc main.c -o main -lrt

#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/wait.h>

int factorial(int n) {
    if (n <= 1)
        return 1;
    return n * factorial(n - 1);
}

int main() {
    int L = 10;
    char memory_name[] = "/my_shared_memory";

    // Create a new shared memory
    int memory_fd = shm_open(memory_name, O_RDWR | O_CREAT | O_EXCL, 0660);
    if(memory_fd == -1) {
        perror("Error when creating shared memory.");
        return -1;
    }

    // Set the size of memory
    ftruncate(memory_fd, L * sizeof(int));
    close(memory_fd);

    for(int l = 0; l < L; l += 1) {
        switch(fork()) {
            case -1:
                perror("Error when calling fork.");
                return -1;
            case 0:
                // Open shared memory in child
                memory_fd = shm_open(memory_name, O_RDWR, 0);
                if(memory_fd == -1){
                    perror("Error when opening shared memory in child.");
                    _exit(-1);
                }

                // Map shared memory in child
                int *data = mmap(NULL, L * sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, memory_fd, 0);
                if(data == MAP_FAILED){
                    perror("Error when mapping shared memory in child.");
                    _exit(-1);
                }

                // Call factorial function
                data[l] = factorial(l + 1);

                // Unmap shared memory
                munmap(data, L * sizeof(int));
                _exit(0);
        }
    }

    // Wait for all children
    for(int l = 0; l < L; l += 1)
        wait(0);


    // Open shared memory in parent
    memory_fd = shm_open(memory_name, O_RDWR, 0);
    if(memory_fd == -1){
        perror("Error when opening shared memory in parent.");
        return -1;
    }

    // Map shared memory in parent
    int *data = mmap(NULL, L * sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, memory_fd, 0);
    if(data == MAP_FAILED){
        perror("Error when mapping shared memory in parent.");
        return -1;
    }

    // Print results
    for(int l = 0; l < L; l++)
        printf("%d ", data[l]);
    printf("\n");

    // Unmap shared memory
    munmap(data, L * sizeof(int));
    // Close file descriptor
    close(memory_fd);
    // Remove shared memory
    shm_unlink(memory_name);

    return 0;
}
```

## Memory mapping - mmap

Memory mapping is generally a two-way mapping of data between memory and file. The data is first loaded from the file into memory. Changes in memory can then be written back to the file.

We use the following functions for memory mappings:

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)
int munmap(void *addr, size_t length)
int msync(void *addr, size_t length, int flags)
```

We create a new mapping with **mmap** function, remove the mapping with **munmap** function, and explicitly write changes back to file with **msync** function. Let's look at the functions and their parameters:

### mmap

The mmap function maps the contents of the file to the selected part of the memory.

The **addr** parameter indicates the beginning of the memory that you would like to initialize. If set to NULL, then the OS will select the appropriate location. Otherwise, the OS will consider our preferences and, depending on the options, map and return the address close to **addr**. In that case the **addr** must be multiple of the page size of the system - the page is the basic memory allocated to the program by the OS.

The **length** parameter defines the length of the mapped memory. With this parameter, we can determine how much of the file is mapped.

The **prot** parameter indicates the protection of the mapped memory. If the program improperly accesses memory (for security reasons), it receives a SIGSEGV signal. Protection can be a combination of the following flags:

- **PROT_NONE** - Pages may not be accessed. This value cannot be combined with others.
- **PROT_READ** - Pages may be read.
- **PROT_WRITE** - Pages may be written.
- **PROT_EXEC** - Pages may be executed.

The **flags** parameter sets different properties of the mapped memory. More important are:

- **MAP_SHARED** - Memory can be shared between different processes. Writing data to the mapped memory with one process is immediately visible to other processes.
- **MAP_PRIVATE** - Mapped memory is visible only to the process that maps it. Initially, different processes still share memory, and the OS makes sure that the changes are visible only to the process that made them.
- **MAP_ANONYMOUS** - We do not use a file for mapping. This flag allows us to reserve new memory with mmap, similar to malloc, calloc, etc. Memory can still be used to communicate between children of this process.
- **MAP_UNINITIALIZED** - The mapped memory is not initialized. When mapping memory with MAP_ANONYMOUS, it is usually initialized to 0. This flag can retrieve uninitialized memory and save some time.

The **fd** parameter is the file descriptor we want to map. This parameter is ignored when using **MAP_ANONYMOUS**.

The **offset** parameter is the first byte of a mapping in a file relative to the beginning of the file. It determines where in the file the mapping starts. Together with length, we can select the specific section of the file that we want to map.

The **function returns** the address of the mapped memory. If we specified the **addr** parameter, then the returned address does not necessarily match the given address. Returns **MAP_FAILED** if an error occurs.

### munmap

The **munmap** function removes the memory mapping.

The **addr** parameter points to the memory where we want to remove the mapping.

The **length** parameter contains the length of memory where the mapping is to be removed.

The memory that we have removed is no longer accessible. We can only remove the mapping of the memory that has been mapped with **mmap**. We can also remove multiple mappings at a time.

The **function returns** 0 on success and -1 on failure.

### msync

The msyc function tells the OS that we want to write changes from the mapped memory back to the file.

The **addr** parameter indicates the memory that we want to write to the file.

The **length** parameter contains the length of memory that we want to write to the file.

The **flags** parameter sets additional options for the **msync** method. It can consist of different values:

- **MS_ASYNC** - The function returns immediately, and the transfer of the data from the memory to the file happens in the background. We cannot combine this flag with **MS_SYNC**.
- **MS_SYNC** - The function waits for the transfer to be completed. We cannot combine this flag with **MS_ASYNC**.
- **MS_INVALIDATE** - The function tells the OS to update other memory mappings of the same file.

The **function returns** 0 on success and -1 on failure.

## Mapping options

There are two types of memory mapping available: **file mapping** and **anonymous mapping (MAP_ANONYMOUS)**. **Mapping a file** to memory maps some or all of the contents of a file. **Anonymous mapping** reserves memory and initializes it to 0. In addition, mapped memory can be private - **MAP_PRIVATE** or shared with other programs - **MAP_SHARED**. There are 4 types of mapped memory from these options. Their use is described below.

### Shared file mapping

The file is loaded into memory and changes in memory are written to the file. All processes that map to the same segment of the file share the same part of the memory. Changes in memory are visible to everyone.

Such a mapping makes it easy to store part of the memory and inter-process communication.

### Private file mapping

The file is loaded into memory. Changes in memory are not written back to the collection. Different processes that map the same part of the collection share the memory until it is changed. Changed parts of memory are accessible only to the process that changes them.

Such mapping makes it easy to initialize memory and save memory. They are also used to map a code running by the same programs - multiple program runs and dynamically linked libraries.

### Shared anonymous mapping

Different programs generally do not have access to the anonymously mapped memory of other programs. However, this type of memory is accessible to threads and split processes. Such a mapping is still suitable for interprocess communication.

### Private anonymous mapping

Mapped memory is available to the current program. Such a mapping is useful to reserve additional memory. The function is internally used by malloc, calloc, ...

## File mapping

Let's take a closer look at the mapping of the file.

We can only map existing data. Therefore, in case the collection does not already exist, we can reserve a certain part of the memory with **ftruncate** function. The following example will create a new file named *text.txt* and reserve enough space for the message. It will then map the collection to memory, copy the message to it, remove the mapping, and close it.

```
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/wait.h>

int main() {
    char text[] = "Lorem Ipsum is simply dummy text of the printing and typesetting industry."
    " Lorem Ipsum has been the industry's standard dummy text ever since the 1500s,"
    " when an unknown printer took a galley of type and scrambled it to make a type specimen book.n";


    int fd = open("text.txt", O_RDWR | O_CREAT, 0770);
    if(ftruncate(fd, strlen(text)) != 0) {
        perror("Error creating a new file.");
        return -1;
    }

    char *text_mmap = mmap(0, strlen(text), PROT_WRITE, MAP_SHARED, fd, 0);
    if(text_mmap == MAP_FAILED)
        perror("Error mapping memory.");
    else
        memcpy(text_mmap, text, strlen(text));

    close(fd);

    if(munmap(text_mmap, strlen(text)) == -1)
        perror("Error removing mapped memory.");

    return 0;
}
```