Lab 6 - Interprocess communication

1. Theory

1.1. Definition

Conventionally, the memory between processes are independent of each other. After Lab 3, we know how a process can transfer its information to other processes with signals, wait(), exit() functions, etc. The main reasons we need processes communicating each other are as follows:

- **Information sharing** Since several applications may be interested in the same piece of information (for instance, copying and pasting), we must provide an environment to allow concurrent access to such information.
- **Computation speedup** If we want a particular task to run faster, we must break it into subtasks, each of which will be executed in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing cores.
- **Modularity** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.

Therefore, we need communication methods to transfer data among processes, which are also known as inter-process communication (IPC) protocols. IPC protocols fundamentally can be categorized into two types, namely shared memory and message passing model. In the shared-memory model, a region of memory that is shared by the cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. This lab will consider two protocols based on this model, namely System V shared memory and memory mapped I/O. Technically, mapped memory is different from System V shared memory but it still has some similarities with the shared-memory model. In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes. Moreover, we will consider two protocols based on the message-passing model, namely pipe, socket.

In this lab, we will explore four more mechanisms to transfer the data between processes, namely:

- Shared memory: Shared memory allows one or more processes to communicate via memory that appears in all of their virtual address spaces. This is also one of the fastest IPC methods and the shared data does not need to be duplicated. The processes can access this memory as quickly as they access their own non-shared memory. The only problem with this mechanism is that the shared memory can become a critical section and therefore, we need a synchronization step to guarantee only a process accessing this section at a particular time. We will go into more detail in the below sections.
- Mapped Memory: Mapped memory permits different processes to communicate via a shared file. Although you can think of mapped memory as using a shared memory

segment with a name, you should be aware that there are technical differences. Mapped memory forms an association between a file and a process's memory. Linux splits the file into page-sized chunks and then copies them into virtual memory pages so that they can be made available in a process's address space. Thus, the process can read the file's contents with ordinary memory access. It can also modify the file's contents by writing to memory. This permits fast access to files. You can think of mapped memory as allocating a buffer to hold a file's entire contents, and then reading the file into the buffer and (if the buffer is modified) writing the buffer back out to the file afterward. Linux handles the file reading and writing operations for you.

- Pipe: A pipe is a communication device that permits unidirectional communication. Data written to the "write end" of the pipe is read back from the "read end." Pipes are serial devices; the data is always read from the pipe in the same order it was written. Typically, a pipe is used to communicate between two threads in a single process or between parent and child processes.
- Socket: A socket is a bidirectional communication device that can be used to communicate with another process on the same machine or with a process running on other machines.

1.2. Shared Memory

1.2.1. Create shared memory segments

A process allocates a shared memory segment using shmget ("SHared Memory GET")

```
int shmget (key_t key , size_t size , int shmflg);
```

- Its first parameter is an integer key that specifies which segment to create and unrelated processes can access the same shared segment by specifying the same key value. Moreover, other processes may have also chosen the same fixed key, which could lead to conflict. So that you should be careful when generating keys for shared memory regions. A solution is that you can use the special constant **IPC_PRIVATE** as the key value guarantees that a brand new memory segment is created.
- Its second parameter specifies the number of bytes in the segment. Because segments are allocated using pages, the number of actually allocated bytes is rounded up to an integral multiple of the page size.
- The third parameter is the bitwise or of flag values that specify options to shmget. The flag values include these:
    - IPC_CREAT: This flag indicates that a new segment should be created. This permits creating a new segment while specifying a key value.

- IPC_EXCL: This flag, which is always used with IPC_CREAT, causes shmget to fail if a segment key is specified that already exists. If this flag is not given and the key of an existing segment is used, shmget returns the existing segment instead of creating a new one.
- Mode flags: This value is made of 9 bits indicating permissions granted to owner, group, and world to control access to the segment.

To make the shared memory segment available/disable, a process must attach/detach it by calling shmat()/shmdt().

```
void *shmat( int shmid , const void *shmaddr , int shmflg ) ;
void *shmdt(const void *shmaddr) ;
```

The shmat() function take three arguments as follows:

- the shared memory segment identifier SHMID returned by shmget().
- a pointer that specifies where in your process's address space you want to map the shared memory; if you specify NULL, Linux will choose an available address. (This argument is also similar to shmdt()).
- The third argument is a flag. You can read more details about this argument in the Linux manual page. (https://man7.org/linux/man-pages/man3/shmat.3p.html).

Examples:
Run the two following processes in two terminals. At the writer process, you can type an input string and observe returns from the reader process.

- writer.cpp

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <cstdio>
#include <unistd.h>
#include <cstdlib>
#define SHM_KEY 0x123
int main(int argc, char *argv[]){
  int shmid;
  char *shm;

  shmid = shmget(SHM_KEY, 1000, 0644 | IPC_CREAT);
  if (shmid < 0) {
    perror("Shared memory");
    return 1;
  }else {
    printf("shared memory: %d\n", shmid);
  }

  shm = (char *)shmat(shmid, 0, 0);
  if (shm == (char *)-1) {
    perror("shmat");
    exit(1);
  }
  sprintf(shm, "hello world\n"); // edit the shared memory.
  printf("shared memory content: %s\n", shm);
  sleep(10);
  // detach from the shared memory
  if (shmdt(shm) == -1) {
    perror("shmdt");
    return 1;
  }
   // Mark the shared segment to be destroyed.
  if (shmctl(shmid, IPC_RMID, 0) == -1) {
    perror("shmctl");
    return 1;
  }
  return 0;
}
```

- reader.cpp

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <cstdio>
#include <cstdlib>

#define SHM_KEY 0x123

int main(int argc, char *argv[]){
  int shmid;
  char *shm;
  shmid = shmget(SHM_KEY, 1000, 0644|IPC_CREAT);
  if (shmid < 0) {
    perror("shmget");
    return 1;
  }
  else {
    printf("shared memory: %d\n", shmid);
  }

  shm = (char *)shmat(shmid, 0,0);
  if (shm == (char *)-1) {
    perror("shmat");
    exit(1);
  }
  printf("shared memory: %p\n", shm);
  if (shm != 0) {
    printf("shared memory content: %s\n", shm);
  }

  sleep(10);
  if (shmdt(shm) == -1) {
    perror("shmdt");
    return 1;
  }
  return 0;
}
```

1.2.2. Synchronization issues in shared memory

In the above programs, when there is only one writer and one reader, the problem may not occur. However, when there are several processes changing the shared memory at the same time, it can lead to **race condition** problems, which will be introduced in more detail in the next lab. In this lab, we will consider a solution for this problem by using **Process Semaphore.** The **semaphore** will lock the shared memory and only allow a process to access this segment at a particular time.

For example, the two following programs will run in parallel. In particular, the ***writer1.c*** should be started first, and it will initialize a variable named ***data***. This variable will contain a counter and a writerID. After both writers are started, they will print the values changed by each other.

- writer1.cpp

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <cstdio>
#include <unistd.h>
```

```
#include <cstdlib>
#include <semaphore.h>
#include <fcntl.h>
#include <stdbool.h>
#include<sys/stat.h>
#define SHM_KEY 0x123
#define SNAME "/semname"
struct shared_data{
    int counter;
    int writerID;
};
int main(int argc, char *argv[]){
  int shmid;
  struct shared_data *data;
  sem_t *sem = sem_open(SNAME, O_CREAT,0644);
  sem_init(sem, 0, 1);
  shmid = shmget(SHM_KEY, 1000, 0644 | IPC_CREAT);
  if (shmid < 0) {
    perror("Shared memory");
    return 1;
  }else {
    printf("shared memory: %d\n", shmid);
  }
  if(sem == SEM_FAILED){
    printf("Sem failed\n");
    return -1;
  }

  data = (struct shared_data *)shmat(shmid, 0,0);
  if (data == (struct shared_data *)-1) {
    perror("shmat");
    exit(1);
  }
  data->counter=0;
  data->writerID=1;

  for(int i=0; i < 20; i++){
    sem_wait(sem);
    printf("Read from Writer ID: %d with counter: %d\n",
data->writerID, data->counter);
```

```
    data->writerID = 1;
    data->counter++;
    sem_post(sem);
    sleep(1);
  }
  if (shmdt(data) == -1) {
    perror("shmdt");
    return 1;
  }

  if (shmctl(shmid, IPC_RMID, 0) == -1) {
    perror("shmctl");
    return 1;
  }
  // Remove the named semaphore
  if (sem_unlink(SNAME) < 0) perror("sem_unlink(3) failed");
  return 0;
}
```

- writer2.cpp

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <cstdio>
#include <unistd.h>
#include <cstdlib>
#include <semaphore.h>
#include <fcntl.h>
#include <stdbool.h>
#include<sys/stat.h>
#define SHM_KEY 0x123
#define SNAME "/semname"
struct shared_data{
    int counter;
    int writerID;
};

int main(int argc, char *argv[]){
  int shmid;
  struct shared_data *data;
  sem_t *sem = sem_open(SNAME, O_CREAT, 0644);
  shmid = shmget(SHM_KEY, 1000, 0644 | IPC_CREAT);
  if (shmid < 0) {
    perror("Shared memory");
    return 1;
  }else {
    printf("shared memory: %d\n", shmid);
  }
  if(sem == SEM_FAILED){
    printf("Sem failed\n");
    return -1;
  }

  data = (struct shared_data *)shmat(shmid, 0,0);
  if (data == (struct shared_data *)-1) {
    perror("shmat");
    exit(1);
  }
  for(int i=0; i < 20; i++){
    sem_wait(sem);
    printf("Read from Writer ID: %d with counter: %d\n",
data->writerID, data->counter);
    data->writerID = 2;
    data->counter++;
    sem_post(sem);
    sleep(1);
  }
  if (shmdt(data) == -1) {
    perror("shmdt");
    return 1;
  }
  sem_close(sem);
  return 0;
}
```

1.3. Mapped Memory

1.3.1. Create a mapped memory

```
#include <sys/mman.h>
void *mmap(void *start, size_t length, int prot, int flags,  int fd,
off_t offset);
```

The ***mmap()*** function asks to map ***length*** bytes starting at offset ***offset*** from the file (or other object) specified by the file descriptor ***fd*** into memory, preferably at address ***start***. If ***start*** is NULL, then the kernel chooses the (page-aligned) address at which to create the mapping; this is the most portable method of creating a new mapping. (In the case that ***start*** is not NULL, you can read more details in *https://man7.org/linux/man-pages/man2/mmap.2.html*).

The ***prot*** argument is used to determine the access permissions of this process to the mapped memory. The available options for ***prot*** are as below.

	Integer Value	**Description**
PROT_READ	1	Read access is allowed.
PROT_WRITE	2	Write access is allowed. Note that this value assumes PROT_READ also.
PROT_NONE	3	No data access is allowed.
PROT_EXEC	4	This value is allowed, but is equivalent to PROT_READ.

The ***flags*** argument is used to control the nature of the map. The following are some common options of ***flags***.

	Description
MAP_SHARED	This flag is used to share the mapping with all other processes, which are mapped to this object. Changes made to the mapping region will be written back to the file.
MAP_PRIVATE	When this flag is used, the mapping will not be seen by any other processes, and the changes made will not be written to the file.
MAP_ANONYMOUS / MAP_ANON	This flag is used to create an anonymous mapping. Anonymous mapping means the mapping is not connected to any files. This mapping is used as the basic primitive to extend the heap.
MAP_FIXED	When this flag is used, the system has to be forced to use the exact mapping address specified in the *address* If this is not possible, then the mapping will fail.

For example,

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

int main(){

    int Number=5;
    int *ptr = mmap(NULL,Number*sizeof(int),
                    PROT_READ | PROT_WRITE,
                    MAP_SHARED | MAP_ANONYMOUS,
                    0,0);

    if(ptr == MAP_FAILED){
        printf("Mapping Failed\n");
        return 1;
    }

    for(int i=0; i < Number; i++){
        ptr[i] = i + 7;
    }

    printf("Initial array's values:");
    for (int i = 0; i < Number; i++ ){
        printf(" %d", ptr[i] );
    }
    printf("\n");
    pid_t child_pid = fork();

    if ( child_pid == 0 ){
        //child
        for (int i = 0; i < Number; i++){
            ptr[i] = ptr[i] * 5;
        }
    }else{
        //parent
        waitpid ( child_pid, NULL, 0);
        printf("\nParent:\n");
        printf("Updated array's values:");
        for (int i = 0; i < Number; i++ ){
           printf(" %d", ptr[i] );
        }
        printf("\n");
    }

    int err = munmap(ptr, Number*sizeof(int));

    if(err != 0){
        printf("Unmapping Failed\n");
        return 1;
    }
    return 0;

}
```

In the above example, we use the MAP_SHARED flag so that the changes in mapped memory will be immediately written to disk. However, if we don't use this flag, the changes may be stored in a buffer before being written to disk. And therefore, we can use the msync() function after updates to force the kernel to write the updates to disk.

```
int msync(void *addr, size_t length, int flags);
```

msync() flushes changes made to the in-core copy of a file that was mapped into memory using ***mmap()*** back to disk. Without use of this call there is no guarantee that changes are written back before ***munmap()*** is called. To be more precise, the part of the file that corresponds to the memory area starting at *addr* and having ***length*** is updated. You can find more information about this function in *https://man7.org/linux/man-pages/man2/msync.2.html*

Question: In the case that there are two or more child processes updating the values in a mapped memory. How can we guarantee this critical section is synchronized?

Answers: We can use the semaphore as in the previous section to guarantee that only a single process can access the critical section.

1.4. Pipe

Pipe actually is a very common method to transfer data between processes. For example, the "pipe" operator '|' can be used to transfer the output from a command to another command as in the following example:

```
# the output from "history" will be input to the grep command.
history | grep "a"
```

The standard library named ***"unistd.h"*** defined the following function to create a pipe. This function creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array ***pipefd*** is used to return two file descriptors referring to the ends of the pipe. ***pipefd[0]*** refers to the read end of the pipe. **pipefd[1]** refers to the write end of the pipe. Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe. In the below program, firstly the parent process will create a pipeline and call fork() to create a child process. Then, the parent process will write a message to the pipeline. At the same time, the child process will read data from the pipeline. Noticeably, both write() and read() need to know the size of the message.

```
#include<stdio.h>
#include<unistd.h>
int main() {
   int pipefds[2];
   int returnstatus;
   int pid;
   char writemessages[20]= "Hello";
   char readmessage[20];
   returnstatus = pipe(pipefds);
   if (returnstatus == -1) {
      printf("Unable to create pipe\n");
      return 1;
   }
   pid = fork();

   // Child process
   if (pid == 0) {
      read(pipefds[0], readmessage, sizeof(readmessage));
      printf("Child Process: Reading, message is %s\n", readmessage);
      return 0;
   }
   //Parent process
   printf("Parent Process: Writing, message is %s\n", writemessages);
   write(pipefds[1], writemessages, sizeof(writemessages));
   return 0;
}
```

Question: So, basically, pipe is a one-way communication protocol, how cam we adapt it for two-way communication? You can answer this question by doing problem 3 in the exercise section.

1.5. Socket

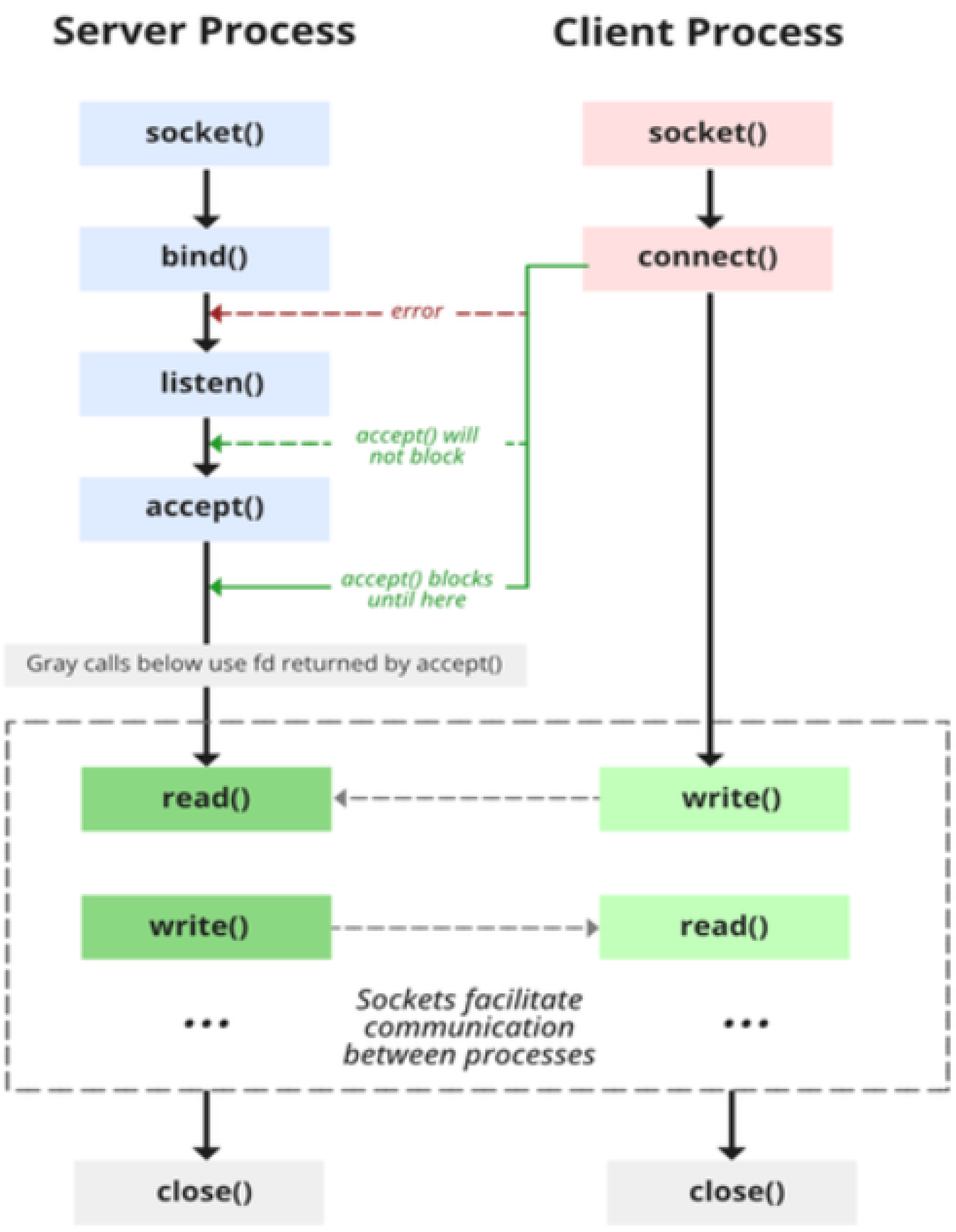

Figure 1.2. State Diagram for server and client model of socket

The above figure depicts the state diagram for server and client model of socket. In particular, both server and client should create a socket object. After that, the server will bind() the socket with an IP, and a port. Then it will listen to the incoming message. The client after initializing the socket it will connect the socket to the remote server and wait unitl the server accepts the connection. After that, they can send/receive messages from/to each other. Unlike pipe, socket is **two-way communication**.

1.5.1. Socket Creation

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

socket() creates an endpoint for communication and returns a file descriptor that refers to that endpoint. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

- The ***domain*** argument specifies a communication domain; this selects the protocol family which will be used for communication. In this lab, we will mainly use **AF_INET** for IPv4 protocol. You can read more about this argument in *https://man7.org/linux/man-pages/man2/socket.2.html*.
- The socket has the indicated ***type***, which specifies the communication semantics. In this lab, we will use **SOCK_STREAM** type as a reliable transmission socket. You can read more about this argument in *https://man7.org/linux/man-pages/man2/socket.2.html*.
- The ***protocol*** specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family, in which case ***protocol*** can be specified as 0.

```
int setsockopt(int socket, int level, int option_name, const void*
option_value, socklen_t option_len);
```

After a socket is created by ***socket()*** call. We can use ***setsockopt()*** to configure some settings for the socket. The ***level*** argument specifies the protocol level at which the option resides. For the sake of simplicity, in this lab, we set options at the socket level, and specify the ***level*** argument as **SOL_SOCKET**. To for the socket to be attached and reuse a IP address and a port, we can use the options named "**SO_REUSEADDR | SO_REUSEPORT**". More details about this options will be illustrated in the example below.

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

After the socket is initialized with configurations, we will bind it with a specific IP address and a particular port by calling bind(). The struct named sockaddr will contain information about this address.

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

This function puts the server socket in a passive mode, where it waits for the client to approach the server to make a connection. The ***backlog*** defines the maximum length to which the queue of pending connections for sockfd may grow. If a connection request arrives when the queue is full, the client may receive an error with an indication of **ECONNREFUSED**.

```
int accept(int sockfd, struct sockaddr *restrict addr, socklen_t
*restrict addrlen);
```

The ***accept()*** call will pop the first connection from the queue of pending connections for the listening socket, ***sockfd***, creates a new connected socket, and returns a new file descriptor referring to that socket. At this point, the connection is established between client and server, and they are ready to transfer data. The original socket ***sockfd*** is unaffected by this call. => we can reuse the ***sockfd*** to accept() the latter connection.
For example,

- server.cpp

```
#include <netinet/in.h>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <sys/socket.h>
#include <unistd.h>
#define PORT 8080
int main(int argc, char const* argv[])
{
    int server_fd, new_socket, valread;
    struct sockaddr_in address;
    int opt = 1;
    int addrlen = sizeof(address);
    char buffer[1024] = { 0 };
    char* hello = "Hello from server";

    // Creating socket file descriptor
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }
    // Forcefully attaching socket to the port 8080
    if (setsockopt(server_fd, SOL_SOCKET,
                   SO_REUSEADDR | SO_REUSEPORT, &opt,
                   sizeof(opt))) {
        perror("setsockopt");
        exit(EXIT_FAILURE);
    }
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    // Forcefully attaching socket to the port 8080
    if (bind(server_fd, (struct sockaddr*)&address,
             sizeof(address))
        < 0) {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }

    while(true){
        if (listen(server_fd, 3) < 0) {
            perror("listen");
            exit(EXIT_FAILURE);
        }
        if ((new_socket = accept(server_fd, (struct
sockaddr*)&address, (socklen_t*)&addrlen)) < 0) {
            perror("accept");
            exit(EXIT_FAILURE);
        }
        valread = read(new_socket, buffer, 1024);
        printf("%s\n", buffer);
        send(new_socket, hello, strlen(hello), 0);
        printf("Hello message sent\n");
       // closing the connected socket
        close(new_socket);
    }
        // closing the listening socket
    shutdown(server_fd, SHUT_RDWR);
    return 0;
}
```

- client.cpp

```
#include <arpa/inet.h>
#include <cstdio>
#include <cstring>
#include <sys/socket.h>
#include <unistd.h>
#define PORT 8080

int main(int argc, char const* argv[])
{
    int sock = 0, valread, client_fd;
    struct sockaddr_in serv_addr;
    char* hello = "Hello from client";
    char buffer[1024] = { 0 };
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        printf("\n Socket creation error \n");
        return -1;
    }

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    // Convert IPv4 and IPv6 addresses from text to binary
    // form
    if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr)
        <= 0) {
        printf(
            "\nInvalid address/ Address not supported \n");
        return -1;
    }

    if ((client_fd
         = connect(sock, (struct sockaddr*)&serv_addr,
                   sizeof(serv_addr)))
        < 0) {
        printf("\nConnection Failed \n");
        return -1;
    }
    send(sock, hello, strlen(hello), 0);
    printf("Hello message sent\n");
    valread = read(sock, buffer, 1024);
    printf("%s\n", buffer);

    // closing the connected socket
    close(client_fd);
    return 0;
}
```

2. Exercises:

1. Firstly, downloading two text files from the url: *https://drive.google.com/file/d/1fgJqOeWbJC4ghMKHkuxfIP6dh2F911-E/view?usp=sharing*

These file contains the 100000 ratings of 943 users for 1682 movies in the following format:

userID <tab> movieID <tab> rating <tab> timeStamp
userID <tab> movieID <tab> rating <tab> timeStamp
...

Secondly, you should write a program that spawns two child processes, and each of them will read a file and compute the average ratings of movies in the file.

a. You implement the program by using the shared memory method. (Note: You should be careful with the synchronization problem).
b. You implement the program by using the mapping memory method. You can use **MAP_ANONYMOUS** to create a no-named mapping memory segment.

2. Conventionally, pipe is a one-way communication method. (In the example at section 1.4, you can test by adding a ***read()*** call after the ***writer()*** call at the parent process, a ***write()*** call after the ***read()*** call at the child process and observe what happens?). However, we still can have some tricks to adapt it for two-way communication by using two pipes. In this exercise, you should implement the TODO segment in the below program.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
static int pipefd1[2],pipefd2[2];

void INIT(void){
    if(pipe(pipefd1)<0 || pipe(pipefd2)<0){
        perror("pipe");
        exit(EXIT_FAILURE);
    }
}
void WRITE_TO_PARENT(void){
  /* send parent a message through pipe*/
   // TO DO
    printf("Child send message to parent!\n");
}
void READ_FROM_PARENT(void){
  /* read message sent by parent from pipe*/
  // TO DO
  printf("Child receive message from parent!\n");
}
void WRITE_TO_CHILD(void){
  /* send child a message through pipe*/
  // TO DO
  printf("Parent send message to child!\n");
}
void READ_FROM_CHILD(void){
  /* read the message sent by child from pipe */
    // TO DO
    printf("Parent receive message from child!\n");
}
int main(int argc, char* argv[]){
    INIT();
    pid_t pid;
    pid = fork();
    //set a timer, process will end after 1 second.
    alarm(10);
    if(pid==0){
        while(1){
            sleep(rand()%2+1);
            WRITE_TO_CHILD();
            READ_FROM_CHILD();
        }
    }else{
        while(1){
            sleep(rand()%2+1);
            READ_FROM_PARENT();
            WRITE_TO_PARENT();
            }
        }
    return 0;
}
```

3. Appendix:

3.1. ipcs command

The ipcs command writes to the standard output information about active interprocess communication facilities. If you do not specify any flags, the ipcs command writes information in a short form about currently active message queues, shared memory segments, semaphores, remote queues, and local queue headers.

- For example, get list of shared memories

```
ipcs -m
```

- For example, get list of semaphore and shared memories

```
ipcs -sm
```

3.2. ipcrm command

ipcrm removes System V inter-process communication (IPC) objects and associated data structures from the system. In order to delete such objects, you must be superuser, or the creator or owner of the object.

For example, remove all IPC resources

```
ipcrm -a
```

For example, remove a specific shared memory

```
ipcrm --shmem-id SHEM_ID
```

3.3. Mapped Map with an Ordinary File

In the case that the cooperating processes do not have parent-child relationships, we can use an ordinary file to share a memory segment of these processes.

For example, the following program ***"mmap-write.cpp"*** opens a file, creating it if it did not previously exist. The third argument to ***open()*** specifies that the file is opened for reading and writing. Because we do not know the file's length, we use ***lseek()*** to ensure that the file is large enough to store an integer and then move back the file position to its beginning. ***lseek()*** call will set the new offset for the file descriptor ***fd.***

The program maps the file and then closes the file descriptor because it's no longer needed. The program then writes a random integer to the mapped memory, and thus the file, and unmaps the memory. The ***munmap()*** call is unnecessary because Linux would automatically unmap the file when the program terminates.

```
// File name: mmap-write.cpp
#include <cstdlib>
#include <cstdio>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <time.h>
#include <unistd.h>
#define FILE_LENGTH 0x100
/* Return a uniformly random number in the range [low,high]. */
int random_range (unsigned const low, unsigned const high)
{
 unsigned const range = high - low + 1;
 return low + (int) (((double) range) * rand () / (RAND_MAX + 1.0));
}
int main (int argc, char* const argv[]){
   int fd;
   void* file_memory;
   /* Seed the random number generator. */
   srand (time (NULL));
   if(argc < 2){
       perror("Not provide file name");
       return 1;
   }
   char *filename = argv[1];
   /* Prepare a file large enough to hold an unsigned integer. */
   fd = open (filename, O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
   // Move the fd to the new offset “FILE_LENGTH+1”
   lseek (fd, FILE_LENGTH+1, SEEK_SET);
   // Write something at the tail of the file to make the file
actually resized.
   write (fd, "", 1);
   // Move the fd back to the head of the file.
   lseek (fd, 0, SEEK_SET);
   /* Create the memory mapping. */
   file_memory = mmap (0, FILE_LENGTH, PROT_WRITE, MAP_SHARED, fd,
0);
   close (fd);
   /* Write a random integer to memory-mapped area. */
   for(int i=0; i<10; i++){
       sprintf((char*) file_memory, "%d\n", random_range (-100,
100));
       sleep(3); // Sleep to wait for mmap-read.cpp
   }
   sprintf((char*) file_memory, "%d\n", -101);

   /* Release the memory (unnecessary because the program exits). */
   munmap (file_memory, FILE_LENGTH);
   return 0;
}
```

You should run the following ***mmap-read.cpp*** after the ***mmap-write.cpp*** and with the same file name.

```
// File name: mmap-read.cpp
#include <cstdlib>
#include <cstdio>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>
int main (int argc, char* const argv[]){
    int fd;
    void* file_memory;
    int integer;
    if(argc < 2){
        perror("Not provide file name");
        return 1;
    }
    char *filename = argv[1];

    /* Open the file. */
    fd = open (argv[1], O_RDWR, S_IRUSR | S_IWUSR);
    struct stat file_info;
    assert(fstat (fd, &file_info) != -1);

    /* Create the memory mapping. */
    file_memory = mmap (0, file_info.st_size, PROT_READ | PROT_WRITE,
 MAP_SHARED, fd, 0);
    close (fd);
    /* Read the integer, print it out, and double it. */
```

```
   for(;; ){
       integer = atoi((char *) file_memory);
       printf ("value: %d\n", integer);
       sleep(3);
       if(integer == -101){
           break;
       }
   }
   /* Release the memory (unnecessary because the program exits). */
   munmap (file_memory, file_info.st_size);
   return 0;
}
```