UNIT-V (SHARED MEMORY AND SOCKETS)

1Q) Describe about Shared Memory?

Shared memory allows one or more processes to communicate via memory that appears in all of their virtual address spaces. The pages of the virtual memory are referenced by page table entries in each of the sharing processes' page tables. It does not have to be at the same address in all of the processes' virtual memory. As with all System V IPC objects, access to shared memory areas is controlled via keys and access rights checking. Once the memory is being shared, there are no checks on how the processes are using it. They must rely on other mechanisms, for example System V semaphores, to synchronize access to the memory.

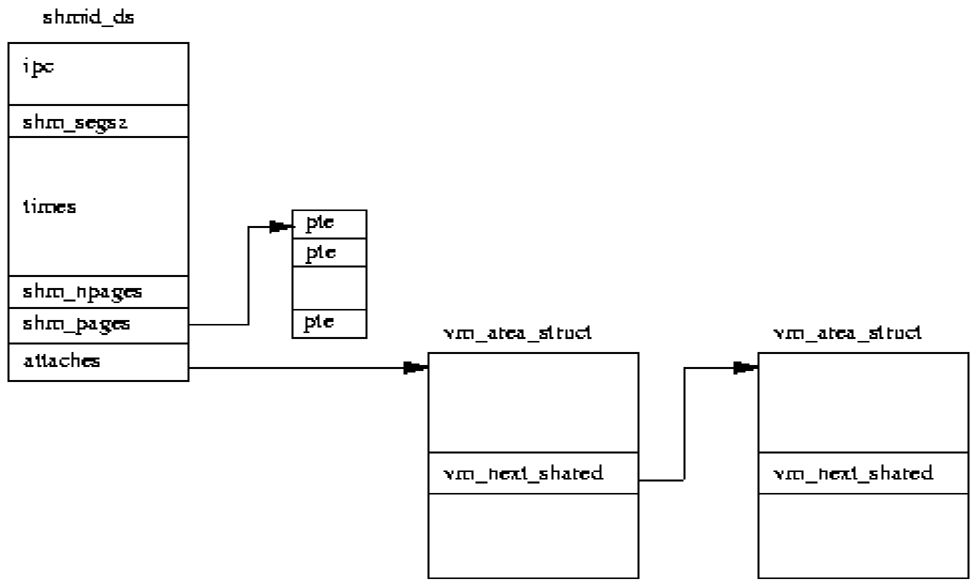


Figure: System V IPC Shared Memory

Each newly created shared memory area is represented by a shmid_ds data structure. These are kept in the shm_segs vector.

The shmid_ds data structure describes how big the area of shared memory is, how many processes are using it and information about how that shared memory is mapped into their address spaces. It is the creator of the shared memory that controls the access permissions to that memory and whether its key is public or private. If it has enough access rights it may also lock the shared memory into physical memory.

Each process that wishes to share the memory must attach to that virtual memory via a system call. This creates a new vm_area_struct data structure describing the shared memory for this process. The process can choose where in its virtual address space the shared memory goes or it can let Linux choose a free area large enough. The new vm_area_struct structure is put into the list of vm_area_struct pointed at by the shmid_ds. The vm_next_shared and vm_prev_shared pointers are used to link them together. The virtual memory is not actually created during the attach; it happens when the first process attempts to access it.

The first time that a process accesses one of the pages of the shared virtual memory, a page fault will occur. When Linux fixes up that page fault it finds the vm_area_struct data structure describing it. This contains pointers to handler routines for this type of shared virtual memory. The shared memory page fault handling code looks in the list of page table entries for this shmid_ds to see if one exists for this page of the shared virtual memory. If it does not exist, it will allocate a physical page and create a page table entry for it. As well as going into the current process's page tables, this entry is saved in the shmid_ds. This means that when the next process that attempts to access this memory gets a page fault, the shared memory fault handling code will use this newly created physical page for that process too. So, the first process that accesses a page of the shared memory causes it to be created and thereafter access by the other processes cause that page to be added into their virtual address spaces.

When processes no longer wish to share the virtual memory, they detach from it. So long as other processes are still using the memory the detach only affects the current process. Its vm_area_struct is removed from the shmid_ds data structure and deallocated. The current process's page tables are updated to invalidate the area of virtual memory that it used to share. When the last process sharing the memory detaches from it, the pages of the shared memory current in physical memory are freed, as is the shmid_ds data structure for this shared memory.

Further complications arise when shared virtual memory is not locked into physical memory. In this case the pages of the shared memory may be swapped out to the system's swap disk during periods of high memory usage.

2Q) Explain the API for shared memory?

Shared Memory Data Structure

```
/* One shmid data structure for each shared memory segment in the system. */

struct shmid_ds {

  struct ipc_perm shm_perm;     /* operation perms */

  int shm_segsz;           /* size of segment (bytes) */

  time_t shm_atime;             /* last attach time */

  time_t shm_dtime;             /* last detach time */

  time_t shm_ctime;             /* last change time */

  unsigned short shm_cpid;      /* pid of creator */

  unsigned short shm_lpid;      /* pid of last operator */

  short shm_nattch;             /* no. of current attaches */

             /* the following are private */

  unsigned short shm_npages;    /* size of segment (pages) */

  unsigned long *shm_pages; /* array of ptrs to frames -> SHMMAX */

  struct vm_area_struct *attaches; /* descriptors for attaches */

};
```

Creating Shared Memory

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

key is either a number or the constant IPC_PRIVATE (man ftok)

a shmid is returned

key_t ftok(const char * path, int id) will return a key value for IPC usage

size is the size of the shared memory data

shmflg is a rights mask (0666) OR'd with one of the following:

IPC_CREAT           will create or attach

IPC_EXCL            creates new or it will error
                    if it exists
```

Attaching to Shared Memory

- After obtaining a shmid from shmget(), you need to *attach* or map the shared memory segment to your data reference:

```
void * shmat(int shmid, void * shmaddr, int shmflg)
```

- shmid is the id returned from shmget()
- shmaddr is the shared memory segment address. Set this to NULL and let the system handle it.
- shmflg is one of the following (usually 0):
  - SHM_RDONLY sets the segment readonly
  - SHM_RND sets page boundary access
  - SHM_SHARE_MMU set first available aligned address

Shared Memory Control

```
struct shmid_ds {

int shm_segsz;                  /* size of segment in bytes */

__time_t shm_atime;             /* time of last shmat command */

__time_t shm_dtime;             /* time of last shmdt command */

...

unsigned short int __shm_npages;      /* size of segment in pages */

msgqnum_t shm_nattach;                /* number of current attaches */

...                       /* pids of creator and last shmop */
};

int shmctl(int shmid, int cmd, struct shmid_ds * buf);

cmd can be one of:

IPC_RMID     destroy the memory specified by shmid

IPC_SETset the uid, gid, and mode of the shared mem

IPC_STAT     get the current shmid_ds struct for the queue

SHM_LOCK Lock the shared memory in memory.

SHM_UNLOCK Unlock the shared memory in memory.
```

3Q) Illustrate with an example to use shmat () & shmdt ()?

```
          #include <sys/ipc.h>
          #include <sys/shm.h>
          #include<unistd.h>
          #include<stdio.h>
          int main()
          {
          int perms=S_IRWXU|S_IRWXG|S_IRWXO;
          int fd=shmget(100,1024,IPC_CREAT|0644);
          If(fd==-1)
          Perror("shmget");
          exit(1);
          char* addr=(char*)shmat(fd,0,0);
          If(addr==(charr*)-1
          Perror("shmat);
          Exit(1);
          Strcpy(addr,"Hello");
          If(shmdt(addr)==-1)
          Perror("shmdt");
          return 0;
          }
```

4Q) Explain about sockets?

A socket acts as an end point in connection between **client** and a **server** present in a network.
Sockets can run on either TCP or UDP protocols
Socket is an interface between application and network.
-The application creates a socket
-The socket *type* dictates the style of communication
End point determined by two things:

- **Host address**: IP address is *Network Layer*
- **Port number**: is *Transport Layer*

• Two end-points determine a connection: socket pair

- ex: 206.62.226.35,p21 + 198.69.10.2,p1500

ex: 206.62.226.35,p21 + 198.69.10.2,p1499
Datagram Socket (UDP)

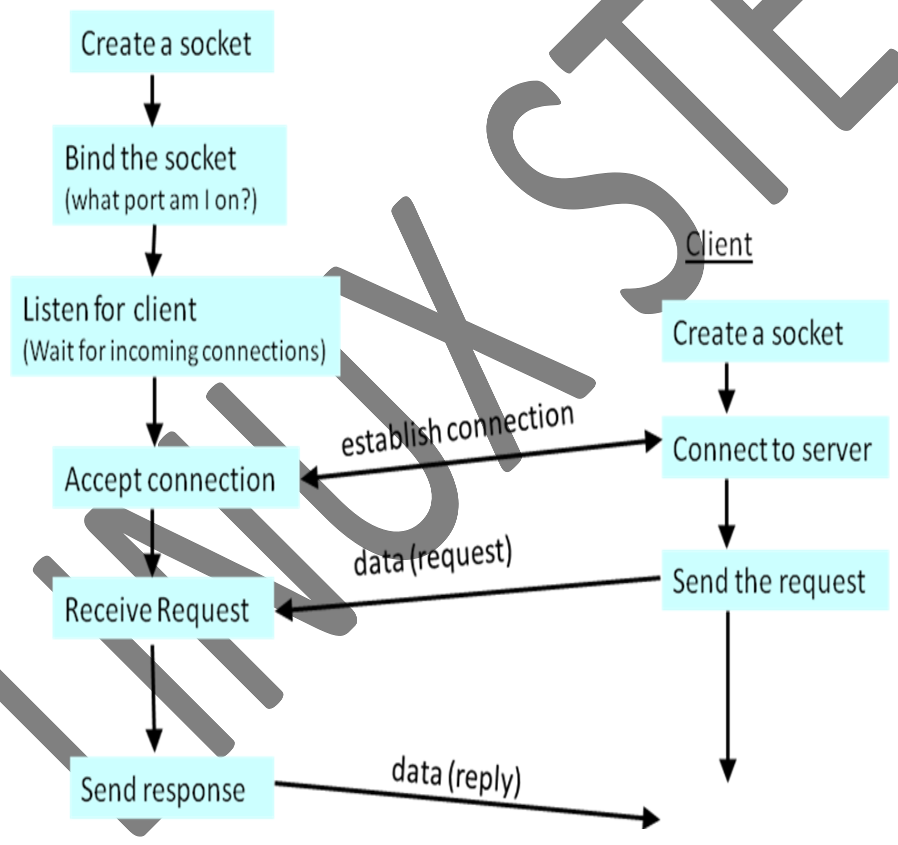
- Collection of messages
- Best effort

- Connectionless: sender or receiver address must be passed along with each message sent from one process to another
- Unreliable
- High speed

Stream Socket (TCP)

- Stream of bytes
- Reliable
- Connection-oriented: sender and receiver socket addresses are pre established before messages are passed between them.
- Low efficiency

5Q) Explain with a neat diagram about Connection Oriented (TCP) Client / server communication using stream sockets?



Socket Address Structure

1. A socket address structure is a special structure that stores the connection details of a socket.
2. It mainly consists of fields like IP address,port number and protocol family.
3. Different protocol suites use different socket address structures.
4. The different socket address structures are
5. IPv4 socket address structure: struct sockaddr_in
6. IPv46 socket address structure: struct sockaddr_in6

7. Generic socket address structure: struct sockaddr

socket(): creating a socket

socket - create an endpoint for communication

The *socket()* function creates an unbound socket in a communications domain, and returns a file descriptor that can be used in later function calls that operate on sockets.

Syntax:

#include <sys/socket.h>

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

The *domain* argument specifies the address family used in the communications domain. The address families supported by the system are implementation-dependent.

The *<sys/socket.h>* header defines at least the following values for the *domain* argument:

AF_UNIX	File system pathnames.
AF_INET	internet address.

The *type* argument specifies the socket type, which determines the semantics of communication over the socket. The socket types supported by the system are implementation-dependent.

Possible socket types include:

SOCK_STREAM : Establishes a virtual circuit for communication. Messages are sent in a sequenced,reliable,

SOCK_DGRAM Provides datagrams, which are connectionless-mode, unreliable messages of fixed maximum length.

SOCK_SEQPACKET Provides sequenced, reliable, bidirectional, connection-mode transmission path for records. A record can be sent using one or more output operations and received using one or more input operations, but a single operation never transfers part of more than one record. Record boundaries are visible to the receiver via the MSG_EOR flag.

Protocol :Specifies a particular protocol to be used with the socket. Specifying a *protocol* of 0 causes *socket()* to use an unspecified default protocol appropriate for the requested socket type.

If the *protocol* argument is non-zero, it must specify a protocol that is supported by the address family. The protocols supported by the system are implementation-dependent.

RETURN VALUE

Upon successful completion, *socket()* returns a nonnegative integer, the socket file descriptor. Otherwise a value of -1 is returned and *errno* is set to indicate the error.

bind()- binds a name to a socket

The *bind()* function assigns an *address* to an unnamed socket. Sockets created with *socket()* function are initially unnamed; they are identified only by their address family.

Syntax:

#include<sys/types.h>

#include <sys/socket.h>

int bind(int *socketid*, const struct sockaddr **address*, socklen_t *address_len*);

The function takes the following arguments:

socketid :Specifies the socket descriptor of the socket to be bound.

address: Points to a sockaddr structure containing the address to be assigned to the socket. The length and format of the address depend on the address family of the socket.

address_len Specifies the length of the sockaddr structure pointed to by the *address* argument.

Upon successful completion, *bind()* returns 0. Otherwise, -1 is returned and *errno* is set to indicate the error

listen() listen for socket connections and limit the queue of incoming connections .

Syntax:
#include<sys/types.h>
#include <sys/socket.h>
int listen(int *socketid*, int *backlog*);

The *listen()* function marks a connection-mode socket, specified by the *socket* argument, as accepting connections, and limits the number of outstanding connections in the socket's listen queue to the value specified by the *backlog* argument.

The socketid argument is a socket descriptor , as returned by a socket sunction call.

The backlog argument specifies the maximum number of connection requests may be queued for the socket.

In most UNIX systems , the maximum allowed value for yhe backlog argument is 5

Upon successful completions, *listen()* returns 0. Otherwise, -1 is returned and *errno* is set to indicate the error.

accept() - accept a new connection on a socket

A server accepts a connection request from a client socket.

This is called in a server process to establish a connection based socket connection with a client socket(which calls *connect* to request connection establishment

Syntax
#include<sys/types.h>
#include <sys/socket.h>
int accept (int *socketid*, struct sockaddr **address*, socklen_t **address_len*);

- The *accept()* function extracts the first connection on the queue of pending connections, creates a new socket with the same socket type protocol and address family as the specified socket, and allocates a new file descriptor for that socket.
- The socketid argument is a socket descriptor , as returned by a socket function call.
- The address argument is a pointer to the address of a socketaddr typed object that holds the name of a client socket where the server socket is connected.
- The address_len argument is initially set to the maximum size of the object pointed to by the address argument.

Connect()-connect a socket

- The *connect()* function requests a connection to be made on a socket.
- A client socket sends a connection request to a server socket.
- Syntax:

#include<sys/types.h>
#include <sys/socket.h>
int connect(int *socketid*, const struct sockaddr **address*, socklen_t *address_len*);

- *socketid* Specifies the file descriptor associated with the socket.
- *address* Points to a sockaddr structure that holds the name of the server socket to be connected.
- *address_len* Specifies the length of the sockaddr structure pointed to by the *address* argument.

send(), sendto() - send a message on a socket

The *send()* function initiates transmission of a message from the specified socket to its peer.

The *send()* function sends a message only when the socket is connected.

Syntax:
#include<sys/types.h>
#include <sys/socket.h>

int send(int *socketid*, const void **buffer*, size_t *length*, int *flags);*

This function sends a message, contained in buffer of size length bytes, to a socket that is connected to the socket, as designated by socketid

sendto() - send a message on a socket

The *sendto()* function sends a message through a connection-mode or connectionless-mode socket. If the socket is connectionless-mode, the message will be sent to the address specified by *dest_addr*. If the socket is connection-mode, *dest_addr* is ignored.

Syntax

#include<sys/types.h>

#include <sys/socket.h>

ssize_t sendto(int *socketid*, const void *buffer, size_t *length*, int *flags*, const struct sockaddr **dest_addr*, socklen_t *dest_len*);

This function is same as the send API , except that the calling process also specifies the address of the reciepent socket name via dest_addr and dest_len.

recv() - receive a message from a connected socket

Syntax:

#include<sys/types.h>

#include <sys/socket.h>

ssize_t recv(int *socketid*, void **buffer*, size_t *length*, int *flags*);

The *recv()* function receives a message from a connection-mode or connectionless-mode socket. It is normally used with connected sockets because it does not permit the application to retrieve the source address of received data.

The recv() function takes the following arguments:

***socketid* Specifies the socket file descriptor.**

buffer Points to a buffer where the message should be stored.

length Specifies the length in bytes of the buffer pointed to by the *buffer* argument.

flags Specifies the type of message reception.

Values of flags argument are formed by logically OR'ing zero or more of the following values:

MSG_PEEK Peeks at an incoming message. The data is treated as unread and the next *recv()* or similar function will still return this data. MSG_OOB Requests out-of-band data. The significance and semantics of out-of-band data are protocol-specific.

MSG_WAITALL Requests that the function block until the full amount of data requested can be returned. The function may return a smaller amount of data if a signal is caught, if the connection is terminated, if MSG_PEEK was specified, or if an error is pending for the socket

recvfrom() - receive a message from a socket

Syntax

#include<sys/types.h>

#include <sys/socket.h>

ssize_t recvfrom(int *socketid*, void **buffer*, size_t *length*, int *flags*, struct sockaddr **address*, socklen_t **address_len*);

The *recvfrom()* function receives a message from a connection-mode or connectionless-mode socket. It is normally used with connectionless-mode sockets because it permits the application to retrieve the source address of received data.

This function is same as the recv API , except that the calling process also specifies the address of the sender socket name via address and address_len.

The function takes the following arguments:

***Socketid* Specifies the socket file descriptor.**

Buffer Points to the buffer where the message should be stored.

length Specifies the length in bytes of the buffer pointed to by the *buffer* argument.

flags Specifies the type of message reception. Values of this argument are formed by logically OR'ing zero or more of the following values:

MSG_PEEK: Peeks at an incoming message. The data is treated as unread and the next *recvfrom()* or similar function will still return this data.

MSG_OOB :Requests out-of-band data. The significance and semantics of out-of-band data are protocol-specific.

MSG_WAITALL: Requests that the function block until the full amount of data requested can be returned. The function may return a smaller amount of data if a signal is caught, if the connection is terminated, if MSG_PEEK was specified, or if an error is pending for the socke

Shutdown ()

- **This function closes the connection between a server and client socket.**

Syntax:

#include<sys/types.h>
#include <sys/socket.h>
int shutdown(int sid,int mode);

The sid argument is a socket descriptor, as returned from a socket function call. This is the socket where the shutdown should occur.

The mode argument specifies the type of shutdown desired. Its possible values and meanings are:

Mode	Meaning
0	Closes the socket for reading. All further reading will return zero bytes (EOF)
1	Closes the socket for writing. Further attempts to send data to the socket will return a -1 failure code.
2	Closes the socket for reading and writing. Further attempts to send data the data to the socket will return a -1 failure code, and any attempt to read data from the socket will receive a zero value (EOF)

6Q) Write a program for Client/server communication using Connection oriented sockets.

In the following example of the server program, the number of incoming connections that the server allows depends on the first parameter that is passed to the server. The default is for the server to allow only one connection.

```
/**** iserver.c ****/
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define SERVER_PORT 12345

/* Run with a number of incoming connection as argument */
int main(int argc, char *argv[])
{
int i, len, num, rc;
int listen_sd, accept_sd;
/* Buffer for data */
char buffer[100];
```

```
struct sockaddr_in addr;

/* If an argument was specified, use it to */
/* control the number of incoming connections */
if(argc >= 2)
num = atoi(argv[1]);
/* Prompt some message */
else
{
printf("Usage: %s <The_number_of_client_connection else 1 will be used>\n",
   argv[0]);
num = 1;
}

/* Create an AF_INET stream socket to receive */
/* incoming connections on */
listen_sd = socket(AF_INET, SOCK_STREAM, 0);
if(listen_sd < 0)
{
perror("Iserver - socket() error");
exit(-1);
}
else
printf("Iserver - socket() is OK\n");

printf("Binding the socket...\n");
/* Bind the socket */
memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = htonl(INADDR_ANY);
addr.sin_port = htons(SERVER_PORT);
rc = bind(listen_sd, (struct sockaddr *)&addr, sizeof(addr));
if(rc < 0)
{
perror("Iserver - bind() error");
close(listen_sd);
exit(-1);
}
else
printf("Iserver - bind() is OK\n");

/* Set the listen backlog */
rc = listen(listen_sd, 5);
if(rc < 0)
{
perror("Iserver - listen() error");
close(listen_sd);
exit(-1);
}
else
printf("Iserver - listen() is OK\n");

/* Inform the user that the server is ready */
```

```
printf("The Iserver is ready!\n");
/* Go through the loop once for each connection */
for(i=0; i < num; i++)
{
/* Wait for an incoming connection */
printf("Iteration: #%d\n", i+1);
printf(" waiting on accept()\n");
accept_sd = accept(listen_sd, NULL, NULL);
if(accept_sd < 0)
{
perror("Iserver - accept() error");
close(listen_sd);
exit(-1);
}
else
printf("accept() is OK and completed successfully!\n");

/* Receive a message from the client */
printf("I am waiting client(s) to send message(s) to me...\n");
rc = recv(accept_sd, buffer, sizeof(buffer), 0);
if(rc <= 0)
{
perror("Iserver - recv() error");
close(listen_sd);
close(accept_sd);
exit(-1);
}
else
printf("The message from client: \"%s\"\n", buffer);
/* Echo the data back to the client */
printf("Echoing it back to client...\n");
len = rc;
rc = send(accept_sd, buffer, len, 0);
if(rc <= 0)
{
perror("Iserver - send() error");
close(listen_sd);
close(accept_sd);
exit(-1);
}
else
printf("Iserver - send() is OK.\n");
/* Close the incoming connection */
close(accept_sd);
}
/* Close the listen socket */
close(listen_sd);
return 0;
}
```

- Compile and link.

```
[bodo@bakawali testsocket]$ gcc -g iserver.c -o iserver
```

- Run the server program.

```
 akawali testsocket]$ ./iserver
```

```
./iserver <The_number_of_client_connection else 1 will be used>
 - socket() is OK
 the socket...
 - bind() is OK
 - listen() is OK
rver is ready!
on: #1
g on accept()
```

- The server is waiting the connections from clients. The following program example is a client program.

: Connection-oriented common client

- This example provides the code for the client job. The client job does a socket (), connect (), send (), recv(), and close().
- The client job is not aware that the data buffer it sent and received is going to a worker job rather than to the server.
- This client job program can also be used to work with other previous connection-oriented server program examples.

```
/****** comclient.c ******/
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
/* Our server port as in the previous program */
#define SERVER_PORT 12345

main (int argc, char *argv[])
{
int len, rc;
int sockfd;
char send_buf[100];
char recv_buf[100];
struct sockaddr_in addr;

if(argc !=2)
{
printf("Usage: %s <Server_name or Server_IP_address>\n", argv[0]);
exit (-1);
}
/* Create an AF_INET stream socket */
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if(sockfd < 0)
{
perror("client - socket() error");
exit(-1);
}
else
printf("client - socket() is OK.\n");
/* Initialize the socket address structure */
memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = htonl(INADDR_ANY);
addr.sin_port = htons(SERVER_PORT);
/* Connect to the server */
rc = connect(sockfd, (struct sockaddr *)&addr, sizeof(struct sockaddr_in));
if(rc < 0)
{
perror("client - connect() error");
close(sockfd);
exit(-1);
}
else
{
printf("client - connect() is OK.\n");
printf("connect() completed successfully.\n");
printf("Connection with %s using port %d established!\n", argv[1], SERVER_PORT);
}

/* Enter data buffer that is to be sent */
printf("Enter message to be sent to server:\n");
gets(send_buf);
/* Send data buffer to the worker job */
len = send(sockfd, send_buf, strlen(send_buf) + 1, 0);
if(len != strlen(send_buf) + 1)
{
perror("client - send() error");
close(sockfd);
exit(-1);
}
else
printf("client - send() is OK.\n");
printf("%d bytes sent.\n", len);
/* Receive data buffer from the worker job */
len = recv(sockfd, recv_buf, sizeof(recv_buf), 0);
if(len != strlen(send_buf) + 1)
{
perror("client - recv() error");
close(sockfd);
exit(-1);
}
else
{
printf("client - recv() is OK.\n");
printf("The sent message: \"%s\" successfully received by server and echoed back
   to client!\n", recv_buf);
printf("%d bytes received.\n", len);
}
/* Close the socket */
close(sockfd);
return 0;
}
```

- Compile and link

```
[bodo@bakawali testsocket]$ gcc -g comclient.c -o comclient
/tmp/ccG1hQSw.o(.text+0x171): In function `main':
/home/bodo/testsocket/comclient.c:53: warning: the `gets' function is dangerous
   and should not be used.
```

- You may want to change the gets() to the secure version, gets_s(). Run the program and make sure you run the server program as in the previous program example.

```
[bodo@bakawali testsocket]$ ./comclient
Usage: ./comclient <Server_name or Server_IP_address>
[bodo@bakawali testsocket]$ ./comclient bakawali
client - socket() is OK.
client - connect() is OK.
connect() completed successfully.
Connection with bakawali using port 12345 established!
Enter message to be sent to server:
This is a test message from a stupid client lol!
client - send() is OK.
49 bytes sent.
client - recv() is OK.
The sent message: "This is a test message from a stupid client lol!" successfully
   received by server and echoed back to client!
49 bytes received.
[bodo@bakawali testsocket]$
```

- And the message at the server console.

```
[bodo@bakawali testsocket]$ ./iserver
Usage: ./iserver <The_number_of_client_connection else 1 will be used>
Iserver - socket() is OK
Binding the socket...
Iserver - bind() is OK
Iserver - listen() is OK
The Iserver is ready!
Iteration: #1
 waiting on accept()
accept() is OK and completed successfully!
I am waiting client(s) to send message(s) to me...
The message from client: "This is a test message from a stupid client lol!"
Echoing it back to client...
Iserver - send() is OK.
[bodo@bakawali testsocket]$
```

- Let try more than 1 connection. Firstly, run the server.

```
[bodo@bakawali testsocket]$ ./iserver 2
Iserver - socket() is OK
Binding the socket...
Iserver - bind() is OK
Iserver - listen() is OK
The Iserver is ready!
Iteration: #1
 waiting on accept()
```

- Then run the client twice.

```
[bodo@bakawali testsocket]$ ./comclient bakawali
client - socket() is OK.
client - connect() is OK.
connect() completed successfully.
Connection with bakawali using port 12345 established!
Enter message to be sent to server:
Test message #1
client - send() is OK.
```

```
16 bytes sent.
client - recv() is OK.
The sent message: "Test message #1" successfully received by server and echoed
   back to client!
16 bytes received.
[bodo@bakawali testsocket]$ ./comclient bakawali
client - socket() is OK.
client - connect() is OK.
connect() completed successfully.
Connection with bakawali using port 12345 established!
Enter message to be sent to server:
Test message #2
client - send() is OK.
16 bytes sent.
client - recv() is OK.
The sent message: "Test message #2" successfully received by server and echoed
   back to client!
16 bytes received.
[bodo@bakawali testsocket]$
```

- The message on the server console.

```
[bodo@bakawali testsocket]$ ./iserver 2
Iserver - socket() is OK
Binding the socket...
Iserver - bind() is OK
Iserver - listen() is OK
The Iserver is ready!
Iteration: #1
 waiting on accept()
accept() is OK and completed successfully!
I am waiting client(s) to send message(s) to me...
The message from client: "Test message #1"
Echoing it back to client...
Iserver - send() is OK.
Iteration: #2
 waiting on accept()
accept() is OK and completed successfully!
I am waiting client(s) to send message(s) to me...
The message from client: "Test message #2"
Echoing it back to client...
Iserver - send() is OK.
[bodo@bakawali test socket]
```

7Q) write a program for Client/server programs using Connectionless sockets?

```
#include <stdio.h>
#include <errno.h>
#include <netinet/in.h>
#define DATA_BUFFER 5000
int main () {
struct sockaddr_in saddr, new_addr;
int fd, ret_val;
char buf[DATA_BUFFER];
socklen_t addrlen;
/* Step1: open a UDP socket */
```

```
    fd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (fd == -1) {
    fprintf(stderr, "socket failed [%s]\n", strerror(errno));
    return -1;
    }
    printf("Created a socket with fd: %d\n", fd);
    /* Initialize the socket address structure */
    saddr.sin_family = AF_INET;
    saddr.sin_port = htons(7000);
    saddr.sin_addr.s_addr = INADDR_ANY;
    /* Step2: bind the socket */
    ret_val = bind(fd, (struct sockaddr *)&saddr, sizeof(struct
sockaddr_in));
    if (ret_val != 0) {
    fprintf(stderr, "bind failed [%s]\n", strerror(errno));
    close(fd);
    return -1;
    }
    /* Step3: Start receiving data. */
    printf("Let us wait for a remote client to send some data\n");
    ret_val = recvfrom(fd, buf, DATA_BUFFER, 0,
    (struct sockaddr *)&new_addr, &addrlen);
    if (ret_val != -1) {
    printf("Received data (len %d bytes): %s\n", ret_val, buf);
    } else {
    printf("recvfrom() failed [%s]\n", strerror(errno));
    }
    /* Last step: close the socket */
    close(fd);
    return 0;
    }

    #include <stdio.h>
    #include <errno.h>
    #include <string.h>
    #include <netinet/in.h>
    #include <netdb.h>
    #define DATA_BUFFER "Mona Lisa was painted by Leonardo da Vinci"
    int main () {
    struct sockaddr_in saddr;
    int fd, ret_val;
    struct hostent *host; /* need netdb.h for this  */
    /* Step1: open a UDP socket */
    fd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (fd == -1) {
    fprintf(stderr, "socket failed [%s]\n", strerror(errno));
    return -1;
    }
    printf("Created a socket with fd: %d\n", fd);
    /* Next, initialize the server address */
    saddr.sin_family = AF_INET;
    saddr.sin_port = htons(7000);
    host = gethostbyname("127.0.0.1");
```

```
        saddr.sin_addr = *((struct in_addr *)host->h_addr);
        /* Step2: send some data */
        ret_val = sendto(fd,DATA_BUFFER, strlen(DATA_BUFFER) + 1, 0,
        (struct sockaddr *)&saddr, sizeof(struct sockaddr_in));
        if (ret_val != -1) {
        printf("Successfully sent data (len %d bytes): %s\n", ret_val,
DATA_BUFFER);
        } else {
        printf("sendto() failed [%s]\n", strerror(errno));
        }
        /* Last step: close the socket */
        close(fd);
        return 0;
        }
        gcc udp-server.c -o udp_server
        $
        $ ./udp_server
        Created a socket with fd: 3
        Let us wait for a remote client to send some data
        Received data (len 43 bytes): Mona Lisa was painted by Leonardo da Vinci
        $ gcc udp-client.c -o udp_client
        $.
        $ ./udp_client
        Created a socket with fd: 3
        Successfully sent data (len 43 bytes): Mona Lisa was painted by Leonardo
da Vinci
```

8Q) Describe how to uses of shared memory?

To use the shared memory, first of all one process should allocate the segment, and then each process desiring to access the segment should attach the segment. After accessing the segment, each process should detach it. It is also necessary to de allocate the segment without fail.
Allocating the shared memory causes virtual pages to be created. It is important to note that allocating the existing segment would not create new pages, but will return the identifier for the existing pages.
All the shared memory segments are allocated as the integral multiples of the system's page size, which is the number of bytes in a page of memory.

9Q) write about UNIX kernel support for shared Memory?

UNIX kernel support for shared memory

There is a shared memory table in the kernel address space that keeps track of all shared memory regions created in the system.
Each entry of the tables stores the following data:

1. Name
2. Creator user ID and group ID.
3. Assigned owner user ID and group ID.
4. Read-write access permission of the region.
5. The time when the last process attached to the region.

6. The time when the last process detached from the region.
7. The time when the last process changed control data of the region.
8. The size, in no. of bytes of the region.

10Q) write about the UNIX API support for shared Memory?

UNIX APIs for shared memory

1) shmget

- Open and create a shared memory.
- Function prototype:

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
int shmget ( key_t key, int size, int flag );
```

- Function returns a positive descriptor if it succeeds or -1 if it fails.

2) Shmat

- Attach a shared memory to a process virtual address space.
- Function prototype:

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
void * shmat ( int shmid, void *addr, int flag );
```

- Function returns the mapped virtual address of he shared memory if it succeeds or -1 if it fails.

3) Shmdt

- Detach a shared memory from the process virtual address space.
- Function prototype:

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
int shmdt ( void *addr );
```

- Function returns 0 if it succeeds or -1 if it fails.

4) Shmctl

- Query or change control data of a shared memory or delete the memory.
- Function prototype:

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
int shmctl ( int shmid, int cmd, struct shmid_ds *buf );
```

- Function returns 0 if it succeeds or -1 if it fails.

11Q) Explain about Berkley Sockets and list the functions or methods provided by the Berkeley sockets API Library?

Berkeley sockets (or BSD sockets) is a computing library with an application programming interface (API) for internet sockets and UNIX domain sockets, used for inter-process communication (IPC).
This list is a summary of functions or methods provided by the Berkeley sockets API library:

1) Socket () creates a new socket of a certain socket type, identified by an integer number, and allocates system resources to it.
2) bind () is typically used on the server side, and associates a socket with a socket address structure, i.e. a specified local port number and IP address.
3) listen() is used on the server side, and causes a bound TCP socket to enter listening state.
4) connect() is used on the client side, and assigns a free local port number to a socket. In case of a TCP socket, it causes an attempt to establish a new TCP connection.
5) accept() is used on the server side. It accepts a received incoming attempt to create a new TCP connection from the remote client, and creates a new socket associated with the socket address pair of this connection.
6) send() and recv(), or write() and read(), or sendto() and recvfrom(), are used for sending and receiving data to/from a remote socket.
7) close() causes the system to release resources allocated to a socket. In case of TCP, the connection is terminated.
8) gethostbyname() and gethostbyaddr() are used to resolve host names and addresses. IPv4 only.
9) select() is used to pend, waiting for one or more of a provided list of sockets to be ready to read, ready to write, or that have errors.
10) poll() is used to check on the state of a socket in a set of sockets. The set can be tested to see if any socket can be written to, read from or if an error occurred.
11) getsockopt() is used to retrieve the current value of a particular socket option for the specified socket.
12) setsockopt() is used to set a particular socket option for the specified socket.

13Q) Explain about IPC over Network using Sockets?

Socket Concepts

When you create a socket, you must specify three parameters:

- Communication style,
- Namespace,
- Protocol.

A communication style controls how the socket treats transmitted data and specifies the number of communication partners. When data is sent through a socket, it is packaged into chunks called *packets*. The communication style determines how these

Packets are handled and how they are addressed from the sender to the receiver.

Connection styles guarantee delivery of all packets in the order they were sent. If packets are lost or reordered by problems in the network, the receiver automatically requests their retransmission from the sender.

A connection-style socket is like a telephone call: The addresses of the sender and receiver are fixed at the beginning of the communication when the connection is established.

Datagram styles do not guarantee delivery or arrival order. Packets may be lost or reordered in transit due to network errors or other conditions. Each packet must be labeled with its destination and is not guaranteed to be delivered. The system guarantees only “best effort,” so packets may disappear or arrive in a different order than shipping.

A datagram-style socket behaves more like postal mail. The sender specifies the receiver’s address for each individual message.

A socket namespace specifies how *socket addresses* are written. A socket address identifies one end of a socket connection. For example, socket addresses in the “local namespace” are ordinary 228

14Q) Explain about UNIX Socket Domain?

We now want to give an example of stream sockets. To do so, we can longer remain in the abstract of general sockets, but we need to pick a domain. We pick the UNIX domain. In the UNIX domain, addresses are pathnames. The corresponding Cstructure is sockaddr_un: struct sockaddr_un {
sa_fami ly_t sun_family ; */* = AF_UNIX */*
char sun_path[108] ; */* socket pathname, NULL☐terminated */*
}
The field sun_path contains a regular pathname, pointing to a special file of type socket (. pipe) which will be created at bind time.
During communication the file will have no content, it is used only as a *rendez-vous* point between processes.

15Q) Explain about Internet Domain Sockets?

UNIX-domain sockets can be used only for communication between two processes on the same computer. *Internet-domain sockets*, on the other hand, may be used to connect processes on different machines connected by a network.

Sockets connecting processes through the Internet use the Internet namespace represented by PF_INET.The most common protocols are TCP/IP.The *Internet Protocol (IP)*, a low-level protocol, moves packets through the Internet, splitting and rejoining the packets, if necessary. It guarantees only “best-effort” delivery, so packets may vanish or be reordered during transport. Every participating computer is specified using a unique IP number.The *Transmission Control Protocol (TCP)*, layered on top of IP, provides reliable connection-ordered transport. It permits telephone-like connections to be established between computers and ensures that data is delivered reliably and inorder.

filenames. In “Internet namespace,” a socket address is composed of the Internet address (also known as an *Internet Protocol address* or *IP address*) of a host attached to the network and a port number.The port number distinguishes among multiple sockets on the same host.

A protocol specifies how data is transmitted. Some protocols are TCP/IP, the primary

networking protocols used by the Internet; the AppleTalk network protocol; and the UNIX local communication protocol. Not all combinations of styles, namespaces,and protocols are supported.

Client-server datagram socket — example

To experiment with datagram sockets in the UNIX domain we will write a client/server application where:

- the client takes a number of arguments on its command line and send them to the server using separate datagrams
- for each datagram received, the server converts it to uppercase and send it back to the client
- the client prints server replies to standard output

For this to work we will need to bind all involved sockets to pathnames.
Client-server datagram socket example — protocol

```
#include <ctype .h>
#include <sys/un.h>
#include <sys/socket .h>
#include <unistd .h>
#include " helpers .h"
#define SRV_SOCK_PATH " /tmp/uc_srv_socket "
#define CLI_SOCK_PATH " /tmp/ uc_cl i_socket .%ld "
#define MSG_LEN 10
#include "uc☐proto .h"
int main( int argc , char *argv [ ] ) { struct sockaddr_un srv_addr , cl i_addr ; int srv_fd , i ;
ssize_t bytes ; socklen_t len ;
char buf [MSG_LEN] ;
i f ( ( srv_fd = socket (AF_UNIX , SOCK_DGRAM, 0) ) < 0) err_sys ( " socket error " ) ;
memset(&srv_addr , 0, sizeof ( struct sockaddr_un ) ) ; srv_addr . sun_family = AF_UNIX ;
strncpy ( srv_addr . sun_path , SRV_SOCK_PATH, sizeof ( srv_addr . sun_path ) ☐ 1) ;
i f ( access ( srv_addr . sun_path , F_OK) == 0) unlink ( srv_addr . sun_path ) ;
i f ( bind ( srv_fd , ( struct sockaddr * ) &srv_addr , sizeof ( struct sockaddr_un ) ) < 0)
err_sys ( " bind error " ) ; 229
```

```
for ( ; ; ) {
len = sizeof ( struct sockaddr_un ) ;
i f ( ( bytes = recvfrom( srv_fd , buf , MSG_LEN, 0, ( struct sockaddr * ) &cl i_addr , &len ) ) <
1) err_sys ( " recvfrom error " ) ;
pr int f ( " server received %ld bytes from %s\n" , ( long) bytes , cl i_addr . sun_path ) ;
for ( i = 0; i < bytes ; i ++)
buf [ i ] = toupper ( ( unsigned char ) buf [ i ] ) ; i f ( sendto ( srv_fd , buf , bytes , 0,
( struct sockaddr * ) &cl i_addr , len ) != bytes ) err_sys ( " sendto error " ) ;
}
}
#include "uc☐proto .h"
int main( int argc , char *argv [ ] ) { struct sockaddr_un srv_addr , cl i_addr ; int srv_fd , i ;
size_t len ; ssize_t bytes ;
char resp [MSG_LEN] ; i f ( argc < 2)
er r_qui t ( "Usage : uc☐c l ient MSG. . . " ) ;
i f ( ( srv_fd = socket (AF_UNIX , SOCK_DGRAM, 0) ) < 0) err_sys ( " socket error " ) ;
memset(&cl i_addr , 0, sizeof ( struct sockaddr_un ) ) ; cl i_addr . sun_family = AF_UNIX ;
snpr int f ( cl i_addr . sun_path , sizeof ( cl i_addr . sun_path ) , CLI_SOCK_PATH, ( long)
getpid ( ) ) ;
i f ( bind ( srv_fd , ( struct sockaddr * ) &cl i_addr , sizeof ( struct sockaddr_un ) ) == ☐1)
err_sys ( " bind error " ) ;
```

Notes:

the server is persistent and processes one datagram at a time, no matter the client rocess, i.e. there is no notion of connection messages larger than 10 bytes are silently truncated