1. How communication happens in shared memory?

ANSWER 1: [Communication in shared memory is a concept where two or more processes can access the same memory region and exchange data or information by reading and writing to that memory1](https://en.wikipedia.org/wiki/Shared_memory)[2](https://www.geeksforgeeks.org/ipc-shared-memory/).

**Shared Memory Communication**

[Shared memory communication is an efficient and fast way of passing data between processes, as it does not involve copying data to and from the kernel space, unlike other methods such as pipes, message queues, etc.1](https://en.wikipedia.org/wiki/Shared_memory)[2](https://www.geeksforgeeks.org/ipc-shared-memory/)

To communicate using shared memory, the processes need to perform the following steps:

* Create or access a shared memory segment: A shared memory segment is a block of memory that can be accessed by multiple processes. The operating system provides system calls to create or access a shared memory segment, such as shmget(), ftok(), etc. [These system calls return an identifier for the shared memory segment, which can be used by the processes to refer to it1](https://en.wikipedia.org/wiki/Shared_memory)[2](https://www.geeksforgeeks.org/ipc-shared-memory/)[3](https://www.ibm.com/docs/en/zos/2.3.0?topic=system-shared-memory-communications).
* Attach or detach from the shared memory segment: Before a process can use a shared memory segment, it needs to attach itself to it using a system call such as shmat(). This system call returns a pointer to the shared memory segment, which can be used by the process to read or write data. When a process is done with the shared memory segment, it should detach itself from it using a system call such as shmdt(). This system call releases the pointer and updates the reference count of the shared memory segment[1](https://en.wikipedia.org/wiki/Shared_memory)[2](https://www.geeksforgeeks.org/ipc-shared-memory/)[3](https://www.ibm.com/docs/en/zos/2.3.0?topic=system-shared-memory-communications).
* Read or write data to the shared memory segment: Once a process is attached to a shared memory segment, it can read or write data to it using normal pointer operations. The data written by one process can be seen by another process that is also attached to the same shared memory segment. However, this also means that there is a possibility of race conditions or data inconsistency if multiple processes try to access the same data simultaneously. Therefore, some synchronization mechanism, such as semaphores, mutexes, etc., may be needed to ensure mutual exclusion and atomicity of operations[1](https://en.wikipedia.org/wiki/Shared_memory)[2](https://www.geeksforgeeks.org/ipc-shared-memory/)[3](https://www.ibm.com/docs/en/zos/2.3.0?topic=system-shared-memory-communications).
* Delete or destroy the shared memory segment: When all the processes are done with the shared memory segment, it should be deleted or destroyed using a system call such as shmctl(). This system call removes the shared memory segment from the system and frees up the resources allocated to it. However, this system call should only be invoked when no process is attached to the shared memory segment, otherwise it may cause errors or undefined behavior[1](https://en.wikipedia.org/wiki/Shared_memory)[2](https://www.geeksforgeeks.org/ipc-shared-memory/)[3](https://www.ibm.com/docs/en/zos/2.3.0?topic=system-shared-memory-communications).

**Example**

Here is an example of how communication happens in shared memory between two processes: a writer process and a reader process. The writer process creates a shared memory segment and writes some data to it. The reader process accesses the same shared memory segment and reads the data from it.

**Writer Process**

#include <stdio.h>

#include <sys/ipc.h>

#include <sys/shm.h>

int main()

{

// Generate a unique key for the shared memory segment

key\_t key = ftok("shmfile", 65);

// Create a shared memory segment with size 1024 bytes

int shmid = shmget(key, 1024, 0666 | IPC\_CREAT);

// Attach to the shared memory segment

char \*data = (char \*)shmat(shmid, (void \*)0, 0);

// Write some data to the shared memory segment

printf("Write Data: ");

fgets(data, 1024, stdin);

printf("Data written in memory: %s\n", data);

// Detach from the shared memory segment

shmdt(data);

return 0;

}

Copy

**Reader Process**

#include <stdio.h>

#include <sys/ipc.h>

#include <sys/shm.h>

int main()

{

// Generate a unique key for the shared memory segment

key\_t key = ftok("shmfile", 65);

// Access the existing shared memory segment with same key

int shmid = shmget(key, 1024, 0666 | IPC\_CREAT);

// Attach to the shared memory segment

char \*data = (char \*)shmat(shmid, (void \*)0, 0);

// Read data from the shared memory segment

printf("Data read from memory: %s\n", data);

// Detach from the shared memory segment

shmdt(data);

// Destroy the shared memory segment

shmctl(shmid, IPC\_RMID, NULL);

return 0;

}

**Output**

Write Data: Hello, this is shared memory communication.

Data written in memory: Hello, this is shared memory communication.

Data read from memory: Hello, this is shared memory communication.

1. Discuss the system call used for reading and writing a pipe.

ANSWER 2: A pipe is a system call that creates a unidirectional communication channel between two processes. [A pipe can be used to send data from one process to another, such as the output of one command to the input of another in a shell1](https://www.geeksforgeeks.org/pipe-system-call/)[2](https://stackoverflow.com/questions/9371401/pipe-system-call).

**Pipe System Call**

The pipe system call is used to create a pipe and returns two file descriptors: one for the read end of the pipe and one for the write end of the pipe. The syntax of the pipe system call in C is:

int pipe(int fds[2]);

Copy

The parameter fds is an array of two integers that will store the file descriptors for the pipe. The first element of the array (fds[0]) will be the file descriptor for the read end of the pipe, and the second element of the array (fds[1](https://www.geeksforgeeks.org/pipe-system-call/)) will be the file descriptor for the write end of the pipe[1](https://www.geeksforgeeks.org/pipe-system-call/)[2](https://stackoverflow.com/questions/9371401/pipe-system-call)[3](https://www.educative.io/answers/how-to-use-the-pipe-system-call-for-inter-process-communication).

The pipe system call returns 0 on success and -1 on error. Some possible errors are:

* EFAULT: The fds parameter is not a valid pointer.
* EMFILE: The process has reached the limit of open file descriptors.
* ENFILE: The system has reached the limit of open file descriptors[1](https://www.geeksforgeeks.org/pipe-system-call/)[2](https://stackoverflow.com/questions/9371401/pipe-system-call)[3](https://www.educative.io/answers/how-to-use-the-pipe-system-call-for-inter-process-communication).

**Reading and Writing a Pipe**

Once a pipe is created, it can be used to send data from one process to another using the read and write system calls. The read system call can be used to read data from the read end of the pipe, and the write system call can be used to write data to the write end of the pipe. The syntax of these system calls in C are:

ssize\_t read(int fd, void \*buf, size\_t count);

ssize\_t write(int fd, const void \*buf, size\_t count);

Copy

The parameters fd, buf and count are similar for both system calls. The fd parameter is the file descriptor for the pipe end (either read or write). The buf parameter is a pointer to a buffer that will store or contain the data to be read or written. The count parameter is the number of bytes to be read or written[1](https://www.geeksforgeeks.org/pipe-system-call/)[2](https://stackoverflow.com/questions/9371401/pipe-system-call)[3](https://www.educative.io/answers/how-to-use-the-pipe-system-call-for-inter-process-communication).

The read and write system calls return the number of bytes actually read or written on success and -1 on error. Some possible errors are:

* EAGAIN: The pipe is empty (for read) or full (for write) and the operation would block.
* EBADF: The fd parameter is not a valid file descriptor or is not open for reading (for read) or writing (for write).
* EFAULT: The buf parameter is not a valid pointer.
* EINTR: The operation was interrupted by a signal.
* EINVAL: The fd parameter is not a valid file descriptor for a pipe.
* EIO: An I/O error occurred[1](https://www.geeksforgeeks.org/pipe-system-call/)[2](https://stackoverflow.com/questions/9371401/pipe-system-call)[3](https://www.educative.io/answers/how-to-use-the-pipe-system-call-for-inter-process-communication).

**Example**

Here is an example of how to use the pipe system call for inter-process communication between a parent process and a child process. The parent process creates a pipe and forks a child process. The parent process writes some data to the write end of the pipe and waits for the child process to terminate. The child process reads the data from the read end of the pipe and prints it to standard output.

#include <stdio.h>

#include <unistd.h>

#include <sys/wait.h>

int main()

{

int fds[2]; // Array to store file descriptors for pipe

char buf[100]; // Buffer to store data

int nbytes; // Number of bytes read or written

// Create a pipe

if (pipe(fds) == -1)

{

perror("pipe");

return 1;

}

// Fork a child process

pid\_t pid = fork();

if (pid == -1)

{

perror("fork");

return 1;

}

else if (pid == 0) // Child process

{

// Close the write end of the pipe

close(fds[1]);

// Read data from the read end of the pipe

nbytes = read(fds[0], buf, sizeof(buf));

if (nbytes == -1)

{

perror("read");

return 1;

}

// Print data to standard output

printf("Child received: %s\n", buf);

// Close the read end of the pipe

close(fds[0]);

}

else // Parent process

{

// Close the read end of the pipe

close(fds[0]);

// Write data to the write end of the pipe

nbytes = write(fds[1], "Hello, child!", 14);

if (nbytes == -1)

{

perror("write");

return 1;

}

// Wait for the child process to terminate

wait(NULL);

// Close the write end of the pipe

close(fds[1]);

}

return 0;

}

The output of this program could look something like this:

Child received: Hello, child!

1. How a cooperating process is varied from an independent process.

ANSWER 3: A cooperating process is a process that can affect or be affected by other processes executing in the system. [A cooperating process can share data or resources with other processes and communicate with them using inter-process communication (IPC) mechanisms1](https://www.javatpoint.com/cooperating-process-in-operating-system)[2](https://www.geeksforgeeks.org/cooperating-process-in-operating-system/).

An independent process is a process that cannot affect or be affected by other processes executing in the system. An independent process does not share any data or resources with other processes and does not communicate with them. An independent process is deterministic, meaning that its output depends only on its input[1](https://www.javatpoint.com/cooperating-process-in-operating-system)[2](https://www.geeksforgeeks.org/cooperating-process-in-operating-system/).

**Differences between Cooperating and Independent Processes**

Some of the differences between cooperating and independent processes are:

* Inter-process communication: Cooperating processes can communicate with each other using IPC mechanisms, such as pipes, message queues, shared memory, semaphores, etc. Independent processes do not communicate with each other at all[1](https://www.javatpoint.com/cooperating-process-in-operating-system)[2](https://www.geeksforgeeks.org/cooperating-process-in-operating-system/).
* Resource sharing: Cooperating processes can share resources, such as CPU, memory, I/O devices, files, etc., with other processes. Independent processes do not share any resources with other processes[1](https://www.javatpoint.com/cooperating-process-in-operating-system)[2](https://www.geeksforgeeks.org/cooperating-process-in-operating-system/).
* Concurrency: Cooperating processes can execute concurrently and interact with each other. Independent processes can execute concurrently but do not interact with each other[1](https://www.javatpoint.com/cooperating-process-in-operating-system)[2](https://www.geeksforgeeks.org/cooperating-process-in-operating-system/).
* Synchronization: Cooperating processes need to synchronize their access to shared resources or data to avoid conflicts or inconsistencies. Independent processes do not need any synchronization as they do not access any shared resources or data[1](https://www.javatpoint.com/cooperating-process-in-operating-system)[2](https://www.geeksforgeeks.org/cooperating-process-in-operating-system/).
* Deadlocks: Cooperating processes may encounter deadlocks, which are situations where a set of processes are waiting for each other to release some resources and none of them can proceed. Independent processes do not encounter deadlocks as they do not wait for any resources held by other processes[1](https://www.javatpoint.com/cooperating-process-in-operating-system)[2](https://www.geeksforgeeks.org/cooperating-process-in-operating-system/).
* Process scheduling: Cooperating processes may affect the process scheduling decisions of the operating system, as they may have different priorities, dependencies, or resource requirements. [Independent processes do not affect the process scheduling decisions of the operating system, as they are treated equally and independently](https://www.javatpoint.com/cooperating-process-in-operating-system)

1. Explain the IPC Model – Pipes with an example.

ANSWER 4: The IPC model – Pipes is a method of inter-process communication (IPC) that allows two or more processes to communicate with each other by creating a unidirectional or bidirectional channel between them. [A pipe is a virtual communication channel that allows data to be transferred between processes, either one-way or two-way1](https://www.geeksforgeeks.org/ipc-technique-pipes/)[2](https://www.tutorialspoint.com/inter_process_communication/inter_process_communication_pipes.htm).

**Pipes**

Pipes are a simple and efficient way for processes to communicate with each other. [Pipes can be used to send data from one process to another, such as the output of one command to the input of another in a shell1](https://www.geeksforgeeks.org/ipc-technique-pipes/)[2](https://www.tutorialspoint.com/inter_process_communication/inter_process_communication_pipes.htm).

To create a pipe, the pipe system call is used, which returns two file descriptors: one for the read end of the pipe and one for the write end of the pipe. The syntax of the pipe system call in C is:

int pipe(int fds[2]);

Copy

The parameter fds is an array of two integers that will store the file descriptors for the pipe. The first element of the array (fds[0]) will be the file descriptor for the read end of the pipe, and the second element of the array (fds[1](https://www.geeksforgeeks.org/ipc-technique-pipes/)) will be the file descriptor for the write end of the pipe[1](https://www.geeksforgeeks.org/ipc-technique-pipes/)[2](https://www.tutorialspoint.com/inter_process_communication/inter_process_communication_pipes.htm)[3](https://www.geeksforgeeks.org/methods-in-interprocess-communication/).

The pipe system call returns 0 on success and -1 on error. Some possible errors are:

* EFAULT: The fds parameter is not a valid pointer.
* EMFILE: The process has reached the limit of open file descriptors.
* ENFILE: The system has reached the limit of open file descriptors[1](https://www.geeksforgeeks.org/ipc-technique-pipes/)[2](https://www.tutorialspoint.com/inter_process_communication/inter_process_communication_pipes.htm)[3](https://www.geeksforgeeks.org/methods-in-interprocess-communication/).

Once a pipe is created, it can be used to send data from one process to another using the read and write system calls. The read system call can be used to read data from the read end of the pipe, and the write system call can be used to write data to the write end of the pipe. The syntax of these system calls in C are:

ssize\_t read(int fd, void \*buf, size\_t count);

ssize\_t write(int fd, const void \*buf, size\_t count);

Copy

The parameters fd, buf and count are similar for both system calls. The fd parameter is the file descriptor for the pipe end (either read or write). The buf parameter is a pointer to a buffer that will store or contain the data to be read or written. The count parameter is the number of bytes to be read or written[1](https://www.geeksforgeeks.org/ipc-technique-pipes/)[2](https://www.tutorialspoint.com/inter_process_communication/inter_process_communication_pipes.htm)[3](https://www.geeksforgeeks.org/methods-in-interprocess-communication/).

The read and write system calls return the number of bytes actually read or written on success and -1 on error. Some possible errors are:

* EAGAIN: The pipe is empty (for read) or full (for write) and the operation would block.
* EBADF: The fd parameter is not a valid file descriptor or is not open for reading (for read) or writing (for write).
* EFAULT: The buf parameter is not a valid pointer.
* EINTR: The operation was interrupted by a signal.
* EINVAL: The fd parameter is not a valid file descriptor for a pipe.
* EIO: An I/O error occurred[1](https://www.geeksforgeeks.org/ipc-technique-pipes/)[2](https://www.tutorialspoint.com/inter_process_communication/inter_process_communication_pipes.htm)[3](https://www.geeksforgeeks.org/methods-in-interprocess-communication/).

**Example**

Here is an example of how to use pipes for IPC between two processes: a parent process and a child process. The parent process creates a pipe and forks a child process. The parent process writes some data to the write end of the pipe and waits for the child process to terminate. The child process reads the data from the read end of the pipe and prints it to standard output.

#include <stdio.h>

#include <unistd.h>

#include <sys/wait.h>

int main()

{

int fds[2]; // Array to store file descriptors for pipe

char buf[100]; // Buffer to store data

int nbytes; // Number of bytes read or written

// Create a pipe

if (pipe(fds) == -1)

{

perror("pipe");

return 1;

}

// Fork a child process

pid\_t pid = fork();

if (pid == -1)

{

perror("fork");

return 1;

}

else if (pid == 0) // Child process

{

// Close the write end of the pipe

close(fds[1]);

// Read data from the read end of the pipe

nbytes = read(fds[0], buf, sizeof(buf));

if (nbytes == -1)

{

perror("read");

return 1;

}

// Print data to standard output

printf("Child received: %s\n", buf);

// Close the read end of the pipe

close(fds[0]);

}

else // Parent process

{

// Close the read end of the pipe

close(fds[0]);

// Write data to the write end of the pipe

nbytes = write(fds[1], "Hello, child!", 14);

if (nbytes == -1)

{

perror("write");

return 1;

}

// Wait for the child process to terminate

wait(NULL);

// Close the write end of the pipe

close(fds[1]);

}

return 0;

}

The output of this program could look something like this:

Child received: Hello, child!

1. Elaborately discuss the various IPC models for Message passing.

ANSWER 5: The IPC models for message passing are methods of inter-process communication (IPC) that allow two or more processes to communicate with each other by exchanging messages without using any shared memory. [Message passing can be used for communication between processes on the same computer or on different computers connected by a network1](https://www.geeksforgeeks.org/inter-process-communication-ipc/)[2](https://eng.libretexts.org/Courses/Delta_College/Operating_System%3A_The_Basics/05%3A_Process_Synchronization/5.4%3A_Interprocess_Communication/5.4.3%3A_IPC_-_Message_Passing_Shared_Memory).

**Message Passing Models**

There are different models of message passing, depending on how the messages are sent and received, how the communication links are established, and how the messages are formatted. Some of the common message passing models are:

* Direct vs. Indirect Communication: In direct communication, the sender and receiver processes need to explicitly name each other in the send and receive operations. This requires prior knowledge of the identities of the communicating processes and a direct link between them. In indirect communication, the sender and receiver processes communicate through an intermediary entity called a mailbox or a port. This allows anonymous and group communication and decouples the sender and receiver processes[1](https://www.geeksforgeeks.org/inter-process-communication-ipc/)[2](https://eng.libretexts.org/Courses/Delta_College/Operating_System%3A_The_Basics/05%3A_Process_Synchronization/5.4%3A_Interprocess_Communication/5.4.3%3A_IPC_-_Message_Passing_Shared_Memory)[3](https://www.guru99.com/inter-process-communication-ipc.html).
* Synchronous vs. Asynchronous Communication: In synchronous communication, the sender and receiver processes are blocked until the message is delivered or received. This ensures that the message is successfully transmitted and acknowledged, but it also introduces delays and dependencies between the communicating processes. In asynchronous communication, the sender and receiver processes are not blocked by the send and receive operations. The sender can continue its execution after sending the message, and the receiver can check for incoming messages at its convenience. This allows more concurrency and flexibility, but it also requires buffering and error handling mechanisms[1](https://www.geeksforgeeks.org/inter-process-communication-ipc/)[2](https://eng.libretexts.org/Courses/Delta_College/Operating_System%3A_The_Basics/05%3A_Process_Synchronization/5.4%3A_Interprocess_Communication/5.4.3%3A_IPC_-_Message_Passing_Shared_Memory)[3](https://www.guru99.com/inter-process-communication-ipc.html).
* Fixed-size vs. Variable-size Messages: In fixed-size message passing, the messages have a predefined length that cannot be exceeded or changed. This simplifies the implementation and allocation of buffers, but it also limits the amount of data that can be transferred in a single message. In variable-size message passing, the messages can have any length up to a maximum limit. This allows more flexibility and efficiency in data transfer, but it also complicates the implementation and allocation of buffers[1](https://www.geeksforgeeks.org/inter-process-communication-ipc/)[2](https://eng.libretexts.org/Courses/Delta_College/Operating_System%3A_The_Basics/05%3A_Process_Synchronization/5.4%3A_Interprocess_Communication/5.4.3%3A_IPC_-_Message_Passing_Shared_Memory)[3](https://www.guru99.com/inter-process-communication-ipc.html).
* Message Format: The message format refers to how the data is organized and encoded in a message. The message format can be binary or text-based, structured or unstructured, human-readable or machine-readable, etc. The message format affects the compatibility, interoperability, security, and performance of message passing systems[1](https://www.geeksforgeeks.org/inter-process-communication-ipc/)[2](https://eng.libretexts.org/Courses/Delta_College/Operating_System%3A_The_Basics/05%3A_Process_Synchronization/5.4%3A_Interprocess_Communication/5.4.3%3A_IPC_-_Message_Passing_Shared_Memory)[3](https://www.guru99.com/inter-process-communication-ipc.html).

**Example**

Here is an example of how to use message passing for IPC between two processes: a client process and a server process. The client process sends a request message to the server process using a socket, which is an endpoint for network communication. The server process receives the request message from the client process using another socket, which is bound to a specific port number. The server process processes the request message and sends a response message back to the client process using the same socket. The client process receives the response message from the server process using its socket.

// Client Process

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#define PORT 8080 // Port number for server

#define MAX 1024 // Maximum size of messages

int main()

{

int sockfd; // Socket file descriptor

char buf[MAX]; // Buffer to store messages

int nbytes; // Number of bytes read or written

struct sockaddr\_in servaddr; // Server address structure

// Create a socket

sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

if (sockfd == -1)

{

perror("socket");

return 1;

}

// Initialize server address structure

memset(&servaddr, 0, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

servaddr.sin\_port = htons(PORT);

servaddr.sin\_addr.s\_addr = inet\_addr("127.0.0.1"); // Localhost IP address

// Connect to server socket

if (connect(sockfd, (struct sockaddr \*)&servaddr, sizeof(servaddr)) == -1)

{

perror("connect");

return 1;

}

// Write request message to server socket

strcpy(buf, "Hello, server!");

nbytes = write(sockfd, buf, strlen(buf) + 1);

if (nbytes == -1)

{

perror("write");

return 1;

}

// Read response message from server socket

nbytes = read(sockfd, buf, sizeof(buf));

if (nbytes == -1)

{

perror("read");

return 1;

}

// Print response message to standard output

printf("Client received: %s\n", buf);

// Close socket

close(sockfd);

return 0;

}

Copy

// Server Process

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#define PORT 8080 // Port number for server

#define MAX 1024 // Maximum size of messages

int main()

{

int sockfd; // Socket file descriptor

int connfd; // Connection file descriptor

char buf[MAX]; // Buffer to store messages

int nbytes; // Number of bytes read or written

struct sockaddr\_in servaddr; // Server address structure

struct sockaddr\_in cliaddr; // Client address structure

socklen\_t len; // Length of client address structure

// Create a socket

sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

if (sockfd == -1)

{

perror("socket");

return 1;

}

// Initialize server address structure

memset(&servaddr, 0, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

servaddr.sin\_port = htons(PORT);

servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY); // Any IP address

// Bind socket to server address

if (bind(sockfd, (struct sockaddr \*)&servaddr, sizeof(servaddr)) == -1)

{

perror("bind");

return 1;

}

// Listen for incoming connections

if (listen(sockfd, 5) == -1)

{

perror("listen");

return 1;

}

// Accept a connection from a client

len = sizeof(cliaddr);

connfd = accept(sockfd, (struct sockaddr \*)&cliaddr, &len);

if (connfd == -1)

{

perror("accept");

return 1;

}

// Read request message from client socket

nbytes = read(connfd, buf, sizeof(buf));

if (nbytes == -1)

{

perror("read");

return 1;

}

// Print request message to standard output

printf("Server received: %s\n", buf);

// Write response message to client socket

strcpy(buf, "Hello, client!");

nbytes = write(connfd, buf, strlen(buf) + 1);

if (nbytes == -1)

{

perror("write");

return 1;

}

// Close connection and socket

close(connfd);

close(sockfd);

return 0;

}

The output of these programs could look something like this:

Server received: Hello, server!

Client received: Hello, client!