1. Write a program to transfer the contents of a file from the parent process to the child process using an unnamed pipe.

Pseudocode

```
BEGIN
   Create a pipe → pipefd[2]
   Fork a child process

   IF process is parent THEN
      Close pipefd[0]  // close read end
      Open the input file
      WHILE not end of file DO
         Read data from file into buffer
         Write buffer into pipefd[1]
      END WHILE
      Close file
      Close pipefd[1]  // writing completed
      Wait for child to finish

   ELSE IF process is child THEN
      Close pipefd[1]  // close write end
      WHILE read(pipefd[0], buffer) > 0 DO
         Print the buffer to stdout (or write to another file)
      END WHILE
      Close pipefd[0]

END
```

Code

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

int main() {
   int pipefd[2];
   char buffer[100];
   pid_t pid;

   // Create pipe
   if (pipe(pipefd) == -1) {
      perror("pipe");
      exit(1);
```

```
    }

    // Create child process
    pid = fork();

    if (pid < 0) {
        perror("fork");
        exit(1);
    }

    if (pid > 0) {
        // -------------------- Parent Process ------------------------
        close(pipefd[0]);  // Close read end

        int fd = open("input.txt", O_RDONLY);
        if (fd < 0) {
            perror("open");
            exit(1);
        }

        int n;
        while ((n = read(fd, buffer, sizeof(buffer))) > 0) {
            write(pipefd[1], buffer, n);   // Send data to child
        }

        close(fd);
        close(pipefd[1]);  // Finished writing
        wait(NULL);        // Wait for child

    } else {
        // -------------------- Child Process ------------------------
        close(pipefd[1]);  // Close write end

        int n;
        while ((n = read(pipefd[0], buffer, sizeof(buffer))) > 0) {
            write(STDOUT_FILENO, buffer, n);  // Print to screen
        }

        close(pipefd[0]);
    }

    return 0;
}
```

2. Implement a program using named pipe (FIFO) to allow one process to send input text and another process to receive and display it.

Pseudocde:

Sender

```
BEGIN
   Create FIFO using mkfifo("myfifo", 0666") if it does not exist
   Open FIFO for writing

   LOOP
      Read input from user
      Write input into FIFO
   END LOOP

   Close FIFO
END
```

Receiver

```
BEGIN
   Open FIFO for reading

   LOOP
      Read data from FIFO
      Display the received text
   END LOOP

   Close FIFO
END
```

Code:

Sender

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

int main() {
   char buffer[100];

   // Create FIFO (will fail if it already exists — ignore error)
```

```
    mkfifo("myfifo", 0666);

    // Open FIFO for writing
    int fd = open("myfifo", O_WRONLY);
    if (fd < 0) {
        perror("open");
        exit(1);
    }

    while (1) {
        printf("Enter message: ");
        fgets(buffer, sizeof(buffer), stdin);

        write(fd, buffer, strlen(buffer) + 1);
    }

    close(fd);
    return 0;
}
```

Receiver

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>

int main() {
    char buffer[100];

    // Open FIFO for reading
    int fd = open("myfifo", O_RDONLY);
    if (fd < 0) {
        perror("open");
        exit(1);
    }

    while (1) {
        int n = read(fd, buffer, sizeof(buffer));
        if (n > 0) {
            printf("Received: %s", buffer);
        }
    }
```

```
    close(fd);
    return 0;
}
```

3. Write a program using a pipe filter that converts text from uppercase to lowercase before printing.

```
BEGIN

    DECLARE integer array pipefd[2]
    DECLARE pid as process ID
    DECLARE buffer[SIZE]

    // 1. Create the unnamed pipe
    CALL pipe(pipefd)
    IF pipe creation fails THEN
        PRINT error and EXIT

    // 2. Create child process
    pid = fork()
    IF fork fails THEN
        PRINT error and EXIT

    // 3. PARENT PROCESS LOGIC
    IF pid > 0 THEN

        CLOSE pipefd[0]      // Close read-end (parent only writes)

        LOOP forever
            DISPLAY "Enter text: "
            READ a line of input into buffer

            WRITE buffer into pipefd[1]
        END LOOP

        CLOSE pipefd[1]

    // 4. CHILD PROCESS LOGIC
    ELSE IF pid == 0 THEN

        CLOSE pipefd[1]      // Close write-end (child only reads)

        LOOP forever
            READ bytes from pipefd[0] into buffer
```

```
        IF read returns 0 (EOF) THEN EXIT loop

        FOR each character in buffer DO
            IF character is between 'A' and 'Z' THEN
                CONVERT char to char + 32 (lowercase)
            END IF
        END FOR

        WRITE the modified buffer to STDOUT
    END LOOP

    CLOSE pipefd[0]

END
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <ctype.h>
#include <string.h>

int main() {
    int pipefd[2];
    char buffer[200];

    // Create pipe
    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(1);
    }

    pid_t pid = fork();

    if (pid < 0) {
        perror("fork");
        exit(1);
    }

    // ---------------------- Parent Process -----------------------
    if (pid > 0) {
        close(pipefd[0]); // Close read end

        while (1) {
```

```
            printf("Enter text: ");
            fgets(buffer, sizeof(buffer), stdin);

            write(pipefd[1], buffer, strlen(buffer) + 1);
        }

        close(pipefd[1]);
    }

    // ---------------------- Child Process ------------------------
    else {
        close(pipefd[1]); // Close write end

        int n;
        while ((n = read(pipefd[0], buffer, sizeof(buffer))) > 0) {
            for (int i = 0; i < n; i++) {
                buffer[i] = tolower(buffer[i]);
            }

            write(STDOUT_FILENO, buffer, n);
        }

        close(pipefd[0]);
    }

    return 0;
}
```

4. Write a program using System V message queues to send and receive messages between two processes.

```
BEGIN

    DEFINE key = 1234               // unique identifier for message queue
    DEFINE msgid as integer
    DECLARE msg structure of type msg_buffer
    DECLARE pid as process ID

    // -------------------------- CREATE QUEUE --------------------------
    CALL msgget(key, IPC_CREAT | 0666) and store result in msgid
    IF msgid == -1 THEN
        PRINT "Error creating message queue"
        EXIT

    // -------------------------- FORK PROCESS --------------------------
```

```
    pid = fork()
    IF pid < 0 THEN
        PRINT "Fork failed"
        EXIT


    // ========================== PARENT PROCESS (SENDER)
==========================
    IF pid > 0 THEN

        LOOP forever
            PRINT "Enter a message: "
            READ input string into msg.mtext

            SET msg.mtype = 1     // message type identifier

            CALL msgsnd(msgid, &msg, sizeof(msg.mtext), 0)
            IF msgsnd returns -1 THEN
                PRINT "Send failed"
            END IF
        END LOOP


    // ========================== CHILD PROCESS (RECEIVER)
==========================
    ELSE

        LOOP forever
            CALL msgrcv(msgid, &msg, sizeof(msg.mtext), 1, 0)

            IF msgrcv returns -1 THEN
                PRINT "Receive failed"
            END IF

            PRINT "Received: " + msg.mtext
        END LOOP

    END IF

END


#include <stdio.h>
#include <stdlib.h>
```

```c
#include <string.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>

// Message structure
struct msg_buffer {
    long mtype;
    char mtext[200];
};

int main() {
    key_t key = 1234;
    int msgid;
    struct msg_buffer msg;

    // Create message queue
    msgid = msgget(key, IPC_CREAT | 0666);
    if (msgid == -1) {
        perror("msgget");
        exit(1);
    }

    pid_t pid = fork();

    if (pid < 0) {
        perror("fork");
        exit(1);
    }

    // ------------------------ Parent Process (Sender) ------------------------
    if (pid > 0) {
        while (1) {
            printf("Enter message: ");
            fgets(msg.mtext, sizeof(msg.mtext), stdin);

            msg.mtype = 1;

            if (msgsnd(msgid, &msg, sizeof(msg.mtext), 0) == -1) {
                perror("msgsnd");
            }
        }
    }
```

```
    // ------------------------ Child Process (Receiver) ------------------------
    else {
        while (1) {
            if (msgrcv(msgid, &msg, sizeof(msg.mtext), 1, 0) == -1) {
                perror("msgrcv");
            }

            printf("Received: %s", msg.mtext);
        }
    }

    return 0;
}
```

5. Implement a program for a chat application between two processes using message queues.

```
BEGIN

    DEFINE a unique key value using ftok() or a constant integer (e.g., 1234)
    DECLARE integer msgid
    DECLARE structure msg of type msg_buffer:
        long mtype
        char mtext[SIZE]

    // ---------------------------- CREATE/ACCESS QUEUE ----------------------------
    CALL msgget(key, IPC_CREAT | 0666) → store result in msgid
    IF msgid == -1 THEN
        PRINT "Error creating message queue"
        EXIT


    // ---------------------------- CHAT LOOP -------------------------------------
    LOOP forever

        // ======================== SENDING LOGIC ==========================
        DISPLAY "You: "
        READ input string into msg.mtext

        SET msg.mtype = SEND_TYPE          // (1 for User1, 2 for User2)

        CALL msgsnd(msgid, &msg, sizeof(msg.mtext), 0)
        IF return value == -1 THEN
            PRINT "Error sending message"
```

```
        // ======================== RECEIVING LOGIC ========================
        CALL msgrcv(msgid, &msg, sizeof(msg.mtext), RECV_TYPE, 0)
          // RECV_TYPE = opposite type (2 for User1, 1 for User2)
        IF return value == -1 THEN
          PRINT "Error receiving message"

        PRINT "Friend: " + msg.mtext

    END LOOP


END
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct msg_buffer {
    long mtype;
    char mtext[200];
};

int main() {
    key_t key = 1234;
    int msgid;
    struct msg_buffer msg;

    // Create or access queue
    msgid = msgget(key, IPC_CREAT | 0666);
    if (msgid == -1) {
        perror("msgget");
        exit(1);
    }

    printf("User1 Chat Started...\n");

    while (1) {
        // ----------- SEND MESSAGE TO USER2 (type = 1) ------------
        printf("You: ");
        fgets(msg.mtext, sizeof(msg.mtext), stdin);
```

```
        msg.mtype = 1;  // user1 sends type 1

        if (msgsnd(msgid, &msg, sizeof(msg.mtext), 0) == -1) {
            perror("msgsnd");
        }

        // ----------- RECEIVE MESSAGE FROM USER2 (type = 2) -------
        if (msgrcv(msgid, &msg, sizeof(msg.mtext), 2, 0) == -1) {
            perror("msgrcv");
        }

        printf("Friend: %s", msg.mtext);
    }

    return 0;
}

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct msg_buffer {
    long mtype;
    char mtext[200];
};

int main() {
    key_t key = 1234;
    int msgid;
    struct msg_buffer msg;

    // Create or access queue
    msgid = msgget(key, IPC_CREAT | 0666);
    if (msgid == -1) {
        perror("msgget");
        exit(1);
    }

    printf("User2 Chat Started...\n");

    while (1) {
```

```
        // ---------- RECEIVE MESSAGE FROM USER1 (type = 1) ----------
        if (msgrcv(msgid, &msg, sizeof(msg.mtext), 1, 0) == -1) {
            perror("msgrcv");
        }

        printf("Friend: %s", msg.mtext);

        // ---------- SEND MESSAGE TO USER1 (type = 2) ---------------
        printf("You: ");
        fgets(msg.mtext, sizeof(msg.mtext), stdin);

        msg.mtype = 2; // user2 sends type 2

        if (msgsnd(msgid, &msg, sizeof(msg.mtext), 0) == -1) {
            perror("msgsnd");
        }
    }

    return 0;
}
```

6. Write a program where one process sends a sequence of numbers using message queues, and another process computes and prints their sum.

```
BEGIN SENDER PROCESS

    DEFINE key = 1234
    DECLARE msgid
    DECLARE structure msg:
        long mtype
        int number

    // -------------------------------- MESSAGE QUEUE SETUP --------------------------------
    CALL msgget(key, IPC_CREAT | 0666) → msgid
    IF msgid == -1 THEN
        PRINT "Message queue error"
        EXIT

    // ----------------------------- SEQUENTIAL NUMBER SENDING LOOP --------------------------
    LOOP forever
        DISPLAY "Enter number (-1 to stop):"
        READ integer into msg.number
```

```
    SET msg.mtype = 1

    CALL msgsnd(msgid, &msg, sizeof(int), 0)
    IF msgsnd == -1 THEN
      PRINT "Error sending number"

    IF msg.number == -1 THEN
      BREAK       // -1 indicates termination
  END LOOP

END SENDER PROCESS


BEGIN RECEIVER PROCESS

  DEFINE key = 1234
  DECLARE msgid
  DECLARE structure msg:
    long mtype
    int number
  DECLARE sum = 0

  // ------------------------------- ACCESS MESSAGE QUEUE -------------------------------
  CALL msgget(key, IPC_CREAT | 0666) → msgid
  IF msgid == -1 THEN
    PRINT "Message queue error"
    EXIT

  // ------------------------------- NUMBER RECEIVING LOOP -------------------------------
  LOOP forever

    CALL msgrcv(msgid, &msg, sizeof(int), 1, 0)
    IF msgrcv == -1 THEN
      PRINT "Error receiving number"

    IF msg.number == -1 THEN
      BREAK        // sender signaled end

    sum = sum + msg.number
    PRINT "Received: ", msg.number, "   Current Sum: ", sum

  END LOOP

  PRINT "Final Sum = ", sum
```

```
    // Optionally delete message queue:
    CALL msgctl(msgid, IPC_RMID, NULL)

END RECEIVER PROCESS
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>

// Message structure
struct msg_buffer {
    long mtype;
    int number;
};

int main() {
    key_t key = 1234;
    int msgid;

    struct msg_buffer msg;

    // Create or access message queue
    msgid = msgget(key, IPC_CREAT | 0666);
    if (msgid == -1) {
        perror("msgget");
        exit(1);
    }

    while (1) {
        printf("Enter number (-1 to stop): ");
        scanf("%d", &msg.number);

        msg.mtype = 1;

        if (msgsnd(msgid, &msg, sizeof(int), 0) == -1) {
            perror("msgsnd");
        }

        if (msg.number == -1) {
            break;
        }
```

```c
    }

    return 0;
}


#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>

// Message structure
struct msg_buffer {
    long mtype;
    int number;
};

int main() {
    key_t key = 1234;
    int msgid;

    struct msg_buffer msg;

    int sum = 0;

    // Create or access message queue
    msgid = msgget(key, IPC_CREAT | 0666);
    if (msgid == -1) {
        perror("msgget");
        exit(1);
    }

    while (1) {
        if (msgrcv(msgid, &msg, sizeof(int), 1, 0) == -1) {
            perror("msgrcv");
        }

        if (msg.number == -1)
            break;

        sum += msg.number;
        printf("Received: %d   Current Sum: %d\n", msg.number, sum);
    }
```

```
    printf("Final Sum = %d\n", sum);

    // Remove message queue
    msgctl(msgid, IPC_RMID, NULL);

    return 0;
}
```

7. Write a program using shared memory (shmget, shmat, shmdt) where the first process writes a string and the second process reads it.

```
BEGIN WRITER PROCESS

    DEFINE key = 1234
    DEFINE shmid as integer
    DEFINE char pointer shm_ptr
    DEFINE BUFFER_SIZE as large enough (e.g., 200)

    // ---------------------------- CREATE SHARED MEMORY SEGMENT --------------------------
    CALL shmget(key, BUFFER_SIZE, IPC_CREAT | 0666) → shmid
    IF shmid == -1 THEN
        PRINT "shmget error"
        EXIT

    // ---------------------------- ATTACH TO THIS SEGMENT -------------------------------
    CALL shmat(shmid, NULL, 0) → shm_ptr
    IF shm_ptr == (void*) -1 THEN
        PRINT "shmat error"
        EXIT

    // ---------------------------- WRITE INTO SHARED MEMORY -----------------------------
    DISPLAY "Enter text:"
    READ a line of text into shm_ptr        // directly writes into shared memory buffer

    // ---------------------------- DETACH FROM SHARED MEMORY ----------------------------
    CALL shmdt(shm_ptr)

END WRITER PROCESS


BEGIN READER PROCESS

    DEFINE key = 1234
    DEFINE shmid as integer
    DEFINE char pointer shm_ptr
```

```
    // ---------------------------- ACCESS SHARED MEMORY SEGMENT --------------------------
    CALL shmget(key, BUFFER_SIZE, 0666) → shmid
    IF shmid == -1 THEN
        PRINT "shmget error"
        EXIT

    // ---------------------------- ATTACH TO THIS SEGMENT -------------------------------
    CALL shmat(shmid, NULL, 0) → shm_ptr
    IF shm_ptr == (void*) -1 THEN
        PRINT "shmat error"
        EXIT

    // ---------------------------- READ AND PRINT DATA ---------------------------------
    PRINT "Received string: ", contents of shm_ptr

    // ---------------------------- DETACH SHARED MEMORY ---------------------------------
    CALL shmdt(shm_ptr)

    // ---------------------------- OPTIONAL: DELETE SHARED MEMORY -----------------------
    CALL shmctl(shmid, IPC_RMID, NULL)

END READER PROCESS
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

#define SIZE 200

int main() {
    key_t key = 1234;
    int shmid;
    char *shm_ptr;

    // Create shared memory segment
    shmid = shmget(key, SIZE, IPC_CREAT | 0666);
    if (shmid == -1) {
        perror("shmget");
        exit(1);
    }
```

```c
    // Attach to shared memory
    shm_ptr = (char *)shmat(shmid, NULL, 0);
    if (shm_ptr == (char *)-1) {
        perror("shmat");
        exit(1);
    }

    printf("Enter a string: ");
    fgets(shm_ptr, SIZE, stdin);  // write directly to shared memory

    // Detach from shared memory
    shmdt(shm_ptr);

    return 0;
}

#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SIZE 200

int main() {
    key_t key = 1234;
    int shmid;
    char *shm_ptr;

    // Access shared memory segment
    shmid = shmget(key, SIZE, 0666);
    if (shmid == -1) {
        perror("shmget");
        exit(1);
    }

    // Attach to shared memory
    shm_ptr = (char *)shmat(shmid, NULL, 0);
    if (shm_ptr == (char *)-1) {
        perror("shmat");
        exit(1);
    }

    printf("Received string: %s", shm_ptr);
```

```
    // Detach from shared memory
    shmdt(shm_ptr);

    // Delete shared memory
    shmctl(shmid, IPC_RMID, NULL);

    return 0;
}
```

8. Implement a shared memory program in which: a. Process 1 takes numbers as input from the user, b. Process 2 sorts the numbers, c. Process 3 displays the sorted list.

```
BEGIN MAIN PROCESS

    DEFINE key = 1234
    DEFINE SIZE = sizeof(shared_data)
    DECLARE shmid
    DECLARE pointer shm_ptr of type shared_data*

    // -------------------------- CREATE SHARED MEMORY ---------------------------
    CALL shmget(key, SIZE, IPC_CREAT | 0666) → shmid
    IF shmid == -1 THEN
        PRINT "Error creating shared memory"
        EXIT

    CALL shmat(shmid, NULL, 0) → shm_ptr
    IF shm_ptr == (void*) -1 THEN
        PRINT "Error attaching shared memory"
        EXIT

    INITIALIZE shm_ptr->status = 0

    // -------------------------- FORK PROCESS 1 --------------------------------
    pid1 = fork()
    IF pid1 == 0 THEN
        // ----------- PROCESS 1: INPUT NUMBERS -----------
        IF shm_ptr->status == 0 THEN
            DISPLAY "Enter number of elements: "
            READ shm_ptr->n
            LOOP i = 0 to n-1
                DISPLAY "Enter element: "
                READ shm_ptr->arr[i]
            END LOOP
```

```
            SET shm_ptr->status = 1     // signal process 2 to sort
        END IF
        EXIT

    ELSE
        // -------------------------- FORK PROCESS 2 --------------------------
        pid2 = fork()
        IF pid2 == 0 THEN
            // ----------- PROCESS 2: SORT NUMBERS -----------
            WAIT until shm_ptr->status == 1

            CALL sorting algorithm (e.g., bubble sort) on shm_ptr->arr[0..n-1]

            SET shm_ptr->status = 2     // signal process 3 to display
            EXIT

        ELSE
            // -------------------------- FORK PROCESS 3 --------------------------
            pid3 = fork()
            IF pid3 == 0 THEN
                // ----------- PROCESS 3: DISPLAY NUMBERS -----------
                WAIT until shm_ptr->status == 2

                DISPLAY "Sorted Array:"
                LOOP i = 0 to shm_ptr->n-1
                    PRINT shm_ptr->arr[i]
                END LOOP

                // clean up shared memory
                CALL shmdt(shm_ptr)
                CALL shmctl(shmid, IPC_RMID, NULL)
                EXIT
            END IF
        END IF
    END IF

    WAIT for all child processes to finish
END
```

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
```

```
#define SIZE 100

// Shared structure
struct shared_data {
    int arr[SIZE];
    int n;
    int status; // 0 = input, 1 = sort, 2 = display
};

int main() {
    key_t key = 1234;
    int shmid;
    struct shared_data *shm_ptr;

    // Create shared memory segment
    shmid = shmget(key, sizeof(struct shared_data), IPC_CREAT | 0666);
    if (shmid == -1) {
        perror("shmget");
        exit(1);
    }

    // Attach to shared memory
    shm_ptr = (struct shared_data *)shmat(shmid, NULL, 0);
    if (shm_ptr == (void *)-1) {
        perror("shmat");
        exit(1);
    }

    shm_ptr->status = 0; // initialize

    pid_t pid1 = fork();

    if (pid1 == 0) {
        // ---------------- PROCESS 1: INPUT ----------------
        printf("Enter number of elements: ");
        scanf("%d", &shm_ptr->n);

        printf("Enter %d numbers:\n", shm_ptr->n);
        for (int i = 0; i < shm_ptr->n; i++) {
            scanf("%d", &shm_ptr->arr[i]);
        }

        shm_ptr->status = 1; // ready for sorting
```

```c
    exit(0);
}

else {
    pid_t pid2 = fork();

    if (pid2 == 0) {
        // ----------------- PROCESS 2: SORT -----------------
        while (shm_ptr->status != 1)
            ; // busy wait until data ready

        // simple bubble sort
        for (int i = 0; i < shm_ptr->n - 1; i++) {
            for (int j = 0; j < shm_ptr->n - i - 1; j++) {
                if (shm_ptr->arr[j] > shm_ptr->arr[j + 1]) {
                    int temp = shm_ptr->arr[j];
                    shm_ptr->arr[j] = shm_ptr->arr[j + 1];
                    shm_ptr->arr[j + 1] = temp;
                }
            }
        }

        shm_ptr->status = 2; // sorted
        exit(0);
    }

    else {
        pid_t pid3 = fork();

        if (pid3 == 0) {
            // ----------------- PROCESS 3: DISPLAY -----------------
            while (shm_ptr->status != 2)
                ; // busy wait until sorted

            printf("Sorted Array: ");
            for (int i = 0; i < shm_ptr->n; i++) {
                printf("%d ", shm_ptr->arr[i]);
            }
            printf("\n");

            // Cleanup shared memory
            shmdt(shm_ptr);
            shmctl(shmid, IPC_RMID, NULL);
            exit(0);
```

```
        }
      }
    }

    // Wait for children to complete
    wait(NULL);
    wait(NULL);
    wait(NULL);

    return 0;
}
```

9. Write a program where one process writes characters A–Z into shared memory, and another process writes the same data into a file.

```
BEGIN PROCESS 1

    CALL shmget(key, size of struct data, IPC_CREAT | 0666) → shmid
    CALL shmat(shmid, NULL, 0) → shm_ptr

    SET shm_ptr->status = 0

    IF shm_ptr->status == 0 THEN
        FOR i = 0 to 25 DO
            shm_ptr->letters[i] = 'A' + i
        END FOR

        SET shm_ptr->status = 1    // signal Process 2
    END IF

    CALL shmdt(shm_ptr)

END

BEGIN PROCESS 2

    CALL shmget(key, size of struct data, 0666) → shmid
    CALL shmat(shmid, NULL, 0) → shm_ptr

    WAIT until shm_ptr->status == 1

    OPEN file "output.txt" for writing

    FOR i = 0 to 25 DO
```

```
        WRITE shm_ptr->letters[i] to file
    END FOR

    CLOSE file

    CALL shmdt(shm_ptr)
    CALL shmctl(shmid, IPC_RMID, NULL)      // delete shared memory

END

#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
#include <fcntl.h>

struct data {
    char letters[26];
    int status;  // 0 = write, 1 = read/write to file
};

int main() {
    key_t key = 1234;
    int shmid;

    struct data *shm_ptr;

    // Create shared memory
    shmid = shmget(key, sizeof(struct data), IPC_CREAT | 0666);
    if (shmid == -1) {
        perror("shmget");
        exit(1);
    }

    // Attach shared memory
    shm_ptr = (struct data *)shmat(shmid, NULL, 0);
    if (shm_ptr == (void *)-1) {
        perror("shmat");
        exit(1);
    }

    shm_ptr->status = 0;
```

```
    pid_t pid = fork();

    if (pid == 0) {
        // ====================== PROCESS 1: Write A-Z ======================
        if (shm_ptr->status == 0) {
            for (int i = 0; i < 26; i++) {
                shm_ptr->letters[i] = 'A' + i;
            }
            shm_ptr->status = 1;  // signal process 2
        }
        shmdt(shm_ptr);
        exit(0);
    }

    else {
        // ====================== PROCESS 2: Write to file ======================
        while (shm_ptr->status != 1)
            ;  // busy wait

        int fd = open("output.txt", O_CREAT | O_WRONLY | O_TRUNC, 0666);
        if (fd < 0) {
            perror("open");
            exit(1);
        }

        write(fd, shm_ptr->letters, 26);
        close(fd);

        printf("Data written to output.txt successfully.\n");

        // cleanup
        shmdt(shm_ptr);
        shmctl(shmid, IPC_RMID, NULL);
    }

    return 0;
}
```

10. Write a program using semaphores to synchronize two processes such that one process prints even numbers and the other prints odd numbers in order.

```
BEGIN MAIN PROCESS

    DEFINE integer LIMIT = N   // total numbers to print
    DECLARE semaphores sem_even, sem_odd
```

```
    // ------------------------- INITIALIZE SEMAPHORES -------------------------
    CALL sem_init(&sem_even, 1, 1)   // even starts first
    CALL sem_init(&sem_odd,  1, 0)   // odd must wait

    // ------------------------- CREATE CHILD PROCESS --------------------------
    pid = fork()

    IF pid == 0 THEN
       // ===================== CHILD PROCESS — PRINT ODD
======================
       FOR i from 1 to LIMIT step 2 DO

          CALL sem_wait(&sem_odd)     // wait for turn

          PRINT i

          CALL sem_post(&sem_even)    // allow even to run
       END FOR

       EXIT CHILD PROCESS


    ELSE
       // ===================== PARENT PROCESS — PRINT EVEN
====================
       FOR i from 0 to LIMIT step 2 DO

          CALL sem_wait(&sem_even)    // wait for turn

          PRINT i

          CALL sem_post(&sem_odd)     // allow odd to run
       END FOR

       WAIT for child to finish
    END IF

    // ------------------------- CLEANUP ---------------------------------------
    CALL sem_destroy(&sem_even)
    CALL sem_destroy(&sem_odd)

END MAIN PROCESS
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <semaphore.h>
#include <fcntl.h>

sem_t sem_even, sem_odd;

int main() {
    int LIMIT = 20;  // print numbers 0 to 20

    // Initialize semaphores
    sem_init(&sem_even, 1, 1);   // even process starts
    sem_init(&sem_odd,  1, 0);   // odd process waits

    pid_t pid = fork();

    if (pid == 0) {
        // ---------------------- CHILD - PRINT ODD ----------------------
        for (int i = 1; i <= LIMIT; i += 2) {

            sem_wait(&sem_odd);
            printf("%d ", i);
            fflush(stdout);

            sem_post(&sem_even);
        }
        exit(0);
    }

    else {
        // ---------------------- PARENT - PRINT EVEN --------------------
        for (int i = 0; i <= LIMIT; i += 2) {

            sem_wait(&sem_even);
            printf("%d ", i);
            fflush(stdout);

            sem_post(&sem_odd);
        }

        wait(NULL);
    }
```

```
    // Cleanup
    sem_destroy(&sem_even);
    sem_destroy(&sem_odd);

    return 0;
}
```

11. Implement the Producer–Consumer problem using shared memory and semaphores.

```
BEGIN MAIN PROCESS

    DEFINE BUFFER_SIZE = N
    CREATE shared memory segment of size shared_data

    ATTACH shared memory → shm_ptr

    INITIALIZE shm_ptr->in = 0
    INITIALIZE shm_ptr->out = 0

    // ---------------- SEMAPHORE INITIALIZATION ----------------
    INIT semaphore mutex = 1
    INIT semaphore empty = BUFFER_SIZE
    INIT semaphore full  = 0

    FORK → producer process
    FORK → consumer process

    WAIT for both children
    DESTROY semaphores
    DETACH & DELETE shared memory

END

BEGIN PRODUCER PROCESS

    LOOP forever (or for fixed items)

        READ an integer item from user

        WAIT(empty)      // wait for empty slot
        WAIT(mutex)      // lock buffer

        INSERT item into buffer[in]
        UPDATE in = (in + 1) mod BUFFER_SIZE
```

```
        SIGNAL(mutex)    // unlock buffer
        SIGNAL(full)     // one more filled slot

    END LOOP

END PRODUCER

BEGIN CONSUMER PROCESS

    LOOP forever (or fixed items)

        WAIT(full)       // wait for item
        WAIT(mutex)      // lock buffer

        REMOVE item from buffer[out]
        UPDATE out = (out + 1) mod BUFFER_SIZE

        SIGNAL(mutex)    // unlock buffer
        SIGNAL(empty)    // one more empty slot

        PRINT item

    END LOOP

END CONSUMER
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>
#include <semaphore.h>

#define SIZE 5   // buffer size

struct shared_data {
    int buffer[SIZE];
    int in;
    int out;
};

sem_t mutex, empty, full;
```

```
int main() {
    key_t key = 1234;
    int shmid;
    struct shared_data *shm_ptr;

    // Create shared memory
    shmid = shmget(key, sizeof(struct shared_data), IPC_CREAT | 0666);
    if (shmid == -1) {
        perror("shmget");
        exit(1);
    }

    // Attach shared memory
    shm_ptr = (struct shared_data *)shmat(shmid, NULL, 0);
    if (shm_ptr == (void *)-1) {
        perror("shmat");
        exit(1);
    }

    // Initialize indices
    shm_ptr->in = 0;
    shm_ptr->out = 0;

    // Initialize semaphores
    sem_init(&mutex, 1, 1);
    sem_init(&empty, 1, SIZE);
    sem_init(&full, 1, 0);

    pid_t pid1 = fork();
    if (pid1 == 0) {
        // ---------------------- PRODUCER ----------------------
        int item;
        while (1) {
            printf("Producer: Enter item: ");
            scanf("%d", &item);

            sem_wait(&empty);
            sem_wait(&mutex);

            shm_ptr->buffer[shm_ptr->in] = item;
            printf("Produced: %d\n", item);
            shm_ptr->in = (shm_ptr->in + 1) % SIZE;
```

```
            sem_post(&mutex);
            sem_post(&full);
        }
        exit(0);
    }

    pid_t pid2 = fork();
    if (pid2 == 0) {
        // ---------------------- CONSUMER ----------------------
        int item;
        while (1) {
            sem_wait(&full);
            sem_wait(&mutex);

            item = shm_ptr->buffer[shm_ptr->out];
            printf("Consumed: %d\n", item);
            shm_ptr->out = (shm_ptr->out + 1) % SIZE;

            sem_post(&mutex);
            sem_post(&empty);

            sleep(1);  // slow down consumer
        }
        exit(0);
    }

    // Parent waits
    wait(NULL);
    wait(NULL);

    // Cleanup
    sem_destroy(&mutex);
    sem_destroy(&empty);
    sem_destroy(&full);

    shmdt(shm_ptr);
    shmctl(shmid, IPC_RMID, NULL);

    return 0;
}
```

12. Implement the Reader–Writer problem using semaphores to control access.
BEGIN MAIN PROCESS

```
    INITIALIZE read_count = 0

    INITIALIZE semaphore mutex = 1
    INITIALIZE semaphore rw_mutex = 1

    FORK → reader process (child 1)
    FORK → writer process (child 2)

    WAIT for both
END

BEGIN READER PROCESS LOOP

    WAIT(mutex)
    INCREMENT read_count

    IF read_count == 1 THEN
        WAIT(rw_mutex)        // first reader blocks writers
    END IF

    SIGNAL(mutex)

    // ---------- CRITICAL SECTION (READING) ----------
    PRINT "Reader is reading data"
    (simulate reading using sleep)

    WAIT(mutex)
    DECREMENT read_count

    IF read_count == 0 THEN
        SIGNAL(rw_mutex)      // last reader releases writer lock
    END IF

    SIGNAL(mutex)

    SLEEP a bit

END LOOP

BEGIN WRITER PROCESS LOOP

    WAIT(rw_mutex)            // writer gets exclusive access

    // --------- CRITICAL SECTION (WRITING) ---------
```

```
    PRINT "Writer is writing data"
    (simulate writing using sleep)

    SIGNAL(rw_mutex)          // release shared resource

    SLEEP a bit

END LOOP

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <semaphore.h>

sem_t mutex, rw_mutex;
int read_count = 0;

void reader() {
    while (1) {

        sem_wait(&mutex);
        read_count++;

        if (read_count == 1) {
            sem_wait(&rw_mutex);  // first reader locks writers
        }
        sem_post(&mutex);

        // ----- CRITICAL SECTION (READING) -----
        printf("Reader: Reading shared data...\n");
        sleep(1);

        sem_wait(&mutex);
        read_count--;

        if (read_count == 0) {
            sem_post(&rw_mutex);  // last reader releases writers
        }
        sem_post(&mutex);

        sleep(1);
    }
}
```

```
void writer() {
    while (1) {

        sem_wait(&rw_mutex);  // exclusive access

        // ----- CRITICAL SECTION (WRITING) -----
        printf("Writer: Writing to shared data...\n");
        sleep(1);

        sem_post(&rw_mutex);  // release exclusive access

        sleep(1);
    }
}

int main() {
    sem_init(&mutex, 1, 1);
    sem_init(&rw_mutex, 1, 1);

    pid_t pid1 = fork();
    if (pid1 == 0) {
        reader();
        exit(0);
    }

    pid_t pid2 = fork();
    if (pid2 == 0) {
        writer();
        exit(0);
    }

    wait(NULL);
    wait(NULL);

    sem_destroy(&mutex);
    sem_destroy(&rw_mutex);

    return 0;
}
```

13. Write a TCP client–server program in C where the client sends a message and the server replies with the reversed string.

```
BEGIN TCP_COMMUNICATION_PROGRAM

    DEFINE SERVER_IP = "127.0.0.1"
    DEFINE PORT = 8080
    DEFINE BUFFER_SIZE = 1024

    // ============================= SERVER SIDE LOGIC
=============================

    SERVER:
        1. CREATE a TCP socket → server_fd = socket(AF_INET, SOCK_STREAM, 0)

        2. INITIALIZE address structure:
            addr.sin_family = AF_INET
            addr.sin_addr.s_addr = INADDR_ANY
            addr.sin_port = htons(PORT)

        3. BIND socket to IP + PORT → bind(server_fd, &addr)

        4. LISTEN for incoming connections → listen(server_fd, backlog=5)

        5. ACCEPT a connection → new_socket = accept(server_fd)

        6. LOOP:
            a. RECEIVE data from client into buffer
            b. COMPUTE reverse string:
                for i from 0 to len/2:
                    swap buffer[i] with buffer[len - i - 1]

            c. SEND reversed string back to client
          END LOOP

        7. CLOSE sockets (new_socket, server_fd)



    // ============================= CLIENT SIDE LOGIC
=============================

    CLIENT:
        1. CREATE a TCP socket → client_fd = socket(AF_INET, SOCK_STREAM, 0)

        2. INITIALIZE server address structure:
            addr.sin_family = AF_INET
```

```
        addr.sin_port = htons(PORT)
        addr.sin_addr = inet_addr(SERVER_IP)

    3. CONNECT to server → connect(client_fd, &addr)

    4. LOOP:
        a. READ a string from USER
        b. SEND the string to server
        c. RECEIVE reversed string from server
        d. DISPLAY the reversed string
      END LOOP

    5. CLOSE client_fd

END
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 8080
#define BUFFER 1024

int main() {
    int server_fd, new_socket;
    char buffer[BUFFER];
    struct sockaddr_in address;
    socklen_t addrlen = sizeof(address);

    // Create socket
    server_fd = socket(AF_INET, SOCK_STREAM, 0);

    // Prepare address
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    // Bind
    bind(server_fd, (struct sockaddr*)&address, sizeof(address));

    // Listen
    listen(server_fd, 5);
```

```
    printf("Server is running... Waiting for connection...\n");

    // Accept connection
    new_socket = accept(server_fd, (struct sockaddr*)&address, &addrlen);
    printf("Client connected.\n");

    while (1) {
        int n = recv(new_socket, buffer, BUFFER, 0);
        if (n <= 0) break;

        buffer[n] = '\0';

        // Reverse the string
        int len = strlen(buffer);
        for (int i = 0; i < len/2; i++) {
            char temp = buffer[i];
            buffer[i] = buffer[len - i - 1];
            buffer[len - i - 1] = temp;
        }

        send(new_socket, buffer, strlen(buffer), 0);
    }

    close(new_socket);
    close(server_fd);
    return 0;
}

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 8080
#define BUFFER 1024

int main() {
    int client_fd;
    char buffer[BUFFER];
    struct sockaddr_in server_addr;

    // Create socket
    client_fd = socket(AF_INET, SOCK_STREAM, 0);
```

```
    // Prepare server address
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");

    // Connect
    connect(client_fd, (struct sockaddr*)&server_addr, sizeof(server_addr));

    while (1) {
        printf("Enter message: ");
        fgets(buffer, BUFFER, stdin);

        send(client_fd, buffer, strlen(buffer), 0);

        int n = recv(client_fd, buffer, BUFFER, 0);
        buffer[n] = '\0';

        printf("Reversed: %s\n", buffer);
    }

    close(client_fd);
    return 0;
}
```

14. Implement a UDP client–server program where the client sends numbers and the server responds with their factorial.

```
BEGIN UDP_FACTORIAL_PROGRAM

    DEFINE SERVER_PORT = 8080
    DEFINE BUFFER_SIZE = 1024

    // ============================= SERVER SIDE
==============================

    SERVER PROCESS:

        1. CREATE UDP socket:
               server_fd = socket(AF_INET, SOCK_DGRAM, 0)

        2. INITIALIZE server_addr:
               server_addr.sin_family = AF_INET
               server_addr.sin_addr.s_addr = INADDR_ANY
               server_addr.sin_port = htons(SERVER_PORT)
```

```
    3. BIND socket to the address

    4. LOOP FOREVER:
        a. RECEIVE integer 'n' from client using recvfrom()
        b. COMPUTE factorial:
             fact = 1
             FOR i from 1 to n:
                fact *= i

        c. SEND factorial result back to client using sendto()

    5. CLOSE socket



  // ============================== CLIENT SIDE ==============================

  CLIENT PROCESS:

    1. CREATE UDP socket:
        client_fd = socket(AF_INET, SOCK_DGRAM, 0)

    2. INITIALIZE server_addr with:
        server IP = "127.0.0.1"
        server port = SERVER_PORT

    3. LOOP FOREVER:
        a. READ integer from user
        b. SEND the integer to server using sendto()
        c. RECEIVE factorial result using recvfrom()
        d. DISPLAY the factorial

    4. CLOSE client_fd

END UDP_FACTORIAL_PROGRAM
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 8080
```

```
long factorial(int n) {
    long fact = 1;
    for (int i = 1; i <= n; i++)
        fact *= i;
    return fact;
}

int main() {
    int server_fd;
    struct sockaddr_in server_addr, client_addr;
    socklen_t len = sizeof(client_addr);
    int number;
    long result;

    // Create UDP socket
    server_fd = socket(AF_INET, SOCK_DGRAM, 0);

    // Setup server address
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    // Bind the socket
    bind(server_fd, (struct sockaddr *)&server_addr, sizeof(server_addr));

    printf("UDP Server running... waiting for numbers...\n");

    while (1) {
        // Receive number
        recvfrom(server_fd, &number, sizeof(number), 0,
                 (struct sockaddr *)&client_addr, &len);

        printf("Received number: %d\n", number);

        // Compute factorial
        result = factorial(number);

        // Send back result
        sendto(server_fd, &result, sizeof(result), 0,
               (struct sockaddr *)&client_addr, len);
    }

    close(server_fd);
    return 0;
```

```c
}

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 8080

int main() {
    int client_fd;
    struct sockaddr_in server_addr;
    int number;
    long result;
    socklen_t len = sizeof(server_addr);

    // Create UDP socket
    client_fd = socket(AF_INET, SOCK_DGRAM, 0);

    // Setup server address
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");

    while (1) {
        printf("Enter a number: ");
        scanf("%d", &number);

        // Send number
        sendto(client_fd, &number, sizeof(number), 0,
               (struct sockaddr *)&server_addr, len);

        // Receive factorial
        recvfrom(client_fd, &result, sizeof(result), 0,
                 (struct sockaddr *)&server_addr, &len);

        printf("Factorial = %ld\n", result);
    }

    close(client_fd);
    return 0;
}
```

15. Write a program to implement an Echo Server using TCP sockets (both iterative and concurrent versions).

```
BEGIN TCP_ECHO_PROGRAM

   DEFINE PORT = 8080
   DEFINE MAX_BUFFER = 1024

   ================================ COMMON SERVER SETUP
================================

   SERVER:
      1. CREATE socket → server_fd = socket(AF_INET, SOCK_STREAM, 0)

      2. CONFIGURE sockaddr_in server_addr:
           sin_family      = AF_INET
           sin_addr.s_addr = INADDR_ANY
           sin_port        = htons(PORT)

      3. BIND server_fd to server_addr

      4. LISTEN(server_fd, backlog=5)


   ================================ ITERATIVE SERVER LOGIC
=============================

      LOOP forever:
         a. ACCEPT one client → new_socket = accept(server_fd)

         b. LOOP:
              i.  RECEIVE data from client
              ii. SEND same data back (echo)
            END LOOP when client disconnects

         c. CLOSE new_socket
      END LOOP



   ================================ CONCURRENT SERVER LOGIC
============================

      LOOP forever:
         a. ACCEPT client → new_socket = accept(server_fd)
```

```
          b. FORK new process

          c. IF child process THEN
               i.   CLOSE server_fd
               ii.  REPEAT:
                      RECEIVE data
                      SEND same data back
                    UNTIL client disconnects
               iii. CLOSE new_socket
               iv.  EXIT child
            ELSE (parent)
               CLOSE new_socket   // parent does not use it
        END LOOP



   =============================== CLIENT LOGIC
=====================================

   CLIENT:
        1. CREATE socket → client_fd

        2. CONFIGURE server_addr with:
             sin_family = AF_INET
             sin_port   = htons(PORT)
             sin_addr   = inet_addr("127.0.0.1")

        3. CONNECT(client_fd, server_addr)

        4. LOOP forever:
             a. READ input string from user
             b. SEND to server
             c. RECEIVE echoed string
             d. PRINT echoed string
        END LOOP

END TCP_ECHO_PROGRAM

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
```

```c
#define PORT 8080
#define BUFFER 1024

int main() {
    int server_fd, new_socket;
    char buffer[BUFFER];
    struct sockaddr_in address;
    socklen_t addrlen = sizeof(address);

    // Create socket
    server_fd = socket(AF_INET, SOCK_STREAM, 0);

    // Address configuration
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    // Bind
    bind(server_fd, (struct sockaddr*)&address, sizeof(address));

    // Listen
    listen(server_fd, 5);

    printf("Iterative Echo Server Running...\n");

    while (1) {
        new_socket = accept(server_fd, (struct sockaddr*)&address, &addrlen);
        printf("Client connected.\n");

        while (1) {
            int n = recv(new_socket, buffer, BUFFER, 0);
            if (n <= 0) break;
            buffer[n] = '\0';
            send(new_socket, buffer, n, 0);
        }

        close(new_socket);
        printf("Client disconnected.\n");
    }

    close(server_fd);
    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/wait.h>

#define PORT 8080
#define BUFFER 1024

int main() {
    int server_fd, new_socket;
    char buffer[BUFFER];
    struct sockaddr_in address;
    socklen_t addrlen = sizeof(address);

    server_fd = socket(AF_INET, SOCK_STREAM, 0);

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    bind(server_fd, (struct sockaddr*)&address, sizeof(address));
    listen(server_fd, 5);

    printf("Concurrent Echo Server Running...\n");

    while (1) {
        new_socket = accept(server_fd, (struct sockaddr*)&address, &addrlen);
        printf("Client connected.\n");

        if (fork() == 0) {
            // Child serves the client
            close(server_fd);

            while (1) {
                int n = recv(new_socket, buffer, BUFFER, 0);
                if (n <= 0) break;
                buffer[n] = '\0';
                send(new_socket, buffer, n, 0);
            }

            close(new_socket);
```

```c
            printf("Client handled in child process.\n");
            exit(0);
        }

        // Parent closes client socket
        close(new_socket);
    }

    close(server_fd);
    return 0;
}
```

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 8080
#define BUFFER 1024

int main() {
    int client_fd;
    char buffer[BUFFER];
    struct sockaddr_in server_addr;

    client_fd = socket(AF_INET, SOCK_STREAM, 0);

    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");

    connect(client_fd, (struct sockaddr*)&server_addr, sizeof(server_addr));

    while (1) {
        printf("Enter message: ");
        fgets(buffer, BUFFER, stdin);

        send(client_fd, buffer, strlen(buffer), 0);

        int n = recv(client_fd, buffer, BUFFER, 0);
        buffer[n] = '\0';

        printf("Echo: %s\n", buffer);
    }
```

```
    close(client_fd);
    return 0;
}
```

16. Job scheduling system: Write a program where a client submits jobs (e.g., factorial, Fibonacci, prime check) via message queues, and a server process executes and returns results.

```
BEGIN JOB_SCHEDULING_SYSTEM_USING_MESSAGE_QUEUES

    DEFINE key = 1234
    DEFINE message queue id = msgid

    DEFINE STRUCT job_request:
        long mtype = 1        // all clients send with type 1
        int  job_type         // 1=factorial, 2=fibonacci, 3=prime
        int  number           // input number
        int  client_id        // client's PID (for response)

    DEFINE STRUCT job_response:
        long mtype = client_id  // response goes only to that client
        long result             // computed result
        int  is_prime_flag      // used only for prime


==========================================================================
======
    SERVER PROCESS LOGIC

==========================================================================
======

    SERVER:
      1. msgid = msgget(key, IPC_CREAT | 0666)

      2. LOOP forever:
          a. RECEIVE job_request where mtype = 1
               msgrcv(msgid, &req, sizeof(req)-sizeof(long), 1, 0)

          b. SWITCH(req.job_type):
                CASE 1: // factorial
                   compute factorial(req.number)
                CASE 2: // fibonacci
                   compute fibonacci(req.number)
```

```
            CASE 3: // prime check
              determine if req.number is prime
          END SWITCH

        c. PREPARE job_response:
            resp.mtype = req.client_id
            resp.result = computed_result
            resp.is_prime_flag = (if prime)

        d. SEND response back to client using:
            msgsnd(msgid, &resp, sizeof(resp)-sizeof(long), 0)
    END LOOP


============================================================================
======
  CLIENT PROCESS LOGIC
============================================================================
======

  CLIENT:
    1. msgid = msgget(key, 0666)
    2. client_pid = getpid()

    LOOP forever:
      a. DISPLAY available job types:
              1 = factorial
              2 = fibonacci
              3 = prime check

      b. READ user’s job_type and number

      c. FORM job_request:
            req.mtype = 1
            req.job_type = job_type
            req.number = number
            req.client_id = client_pid

      d. SEND request:
            msgsnd(msgid, &req, sizeof(req)-sizeof(long), 0)

      e. WAIT for response:
```

```
            msgrcv(msgid, &resp, sizeof(resp)-sizeof(long),
                client_pid, 0)

        f. DISPLAY result:
            IF job_type == 3: print prime/not prime
            ELSE: print numeric result
    END LOOP

END JOB_SCHEDULING_SYSTEM
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <unistd.h>

#define KEY 1234

struct job_request {
    long mtype;
    int job_type;
    int number;
    int client_id;
};

struct job_response {
    long mtype;
    long result;
    int is_prime;
};

// factorial
long factorial(int n) {
    long f = 1;
    for (int i = 1; i <= n; i++) f *= i;
    return f;
}

// fibonacci
long fibonacci(int n) {
    if (n <= 1) return n;
    long a = 0, b = 1, c;
    for (int i = 2; i <= n; i++) {
        c = a + b;
```

```c
        a = b;
        b = c;
    }
    return b;
}

// prime check
int is_prime(int n) {
    if (n <= 1) return 0;
    for (int i = 2; i*i <= n; i++)
        if (n % i == 0) return 0;
    return 1;
}

int main() {
    int msgid = msgget(KEY, IPC_CREAT | 0666);
    struct job_request req;
    struct job_response resp;

    printf("Server started...\n");

    while (1) {
        // receive request
        msgrcv(msgid, &req, sizeof(req)-sizeof(long), 1, 0);

        long result = 0;
        int prime = 0;

        switch (req.job_type) {
            case 1: result = factorial(req.number); break;
            case 2: result = fibonacci(req.number); break;
            case 3: prime = is_prime(req.number); break;
        }

        // prepare response
        resp.mtype = req.client_id;
        resp.result = result;
        resp.is_prime = prime;

        // send response
        msgsnd(msgid, &resp, sizeof(resp)-sizeof(long), 0);
    }

    return 0;
```

```c
}

#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <unistd.h>

#define KEY 1234

struct job_request {
    long mtype;
    int job_type;
    int number;
    int client_id;
};

struct job_response {
    long mtype;
    long result;
    int is_prime;
};

int main() {
    int msgid = msgget(KEY, 0666);
    struct job_request req;
    struct job_response resp;

    int client_pid = getpid();
    printf("Client PID = %d\n", client_pid);

    while (1) {
        printf("\n--- Job Menu ---\n");
        printf("1. Factorial\n");
        printf("2. Fibonacci\n");
        printf("3. Prime Check\n");
        printf("Enter job type: ");
        scanf("%d", &req.job_type);

        printf("Enter number: ");
        scanf("%d", &req.number);

        req.mtype = 1;
        req.client_id = client_pid;
```

```
        // send request
        msgsnd(msgid, &req, sizeof(req)-sizeof(long), 0);

        // wait for response
        msgrcv(msgid, &resp, sizeof(resp)-sizeof(long), client_pid, 0);

        // print result
        if (req.job_type == 3) {
            if (resp.is_prime)
                printf("Result: %d is PRIME\n", req.number);
            else
                printf("Result: %d is NOT prime\n", req.number);
        } else {
            printf("Result: %ld\n", resp.result);
        }
    }

    return 0;
}
```

17. Priority-based messaging: Implement a system where multiple clients send messages with different priorities, and the server displays them in priority order.

```
BEGIN PRIORITY_BASED_MESSAGE_SYSTEM

    DEFINE KEY = 1234

    DEFINE STRUCT message:
            long mtype        // priority: higher value = higher priority
            char text[200]
            int client_id     // optional (for identification)


    ========================= SERVER PROCESS ==========================

    SERVER:
        1. CREATE/GET message queue:
                msgid = msgget(KEY, IPC_CREAT | 0666)

        2. LOOP FOREVER:
                a. RECEIVE highest priority message:
                        msgrcv(msgid, &msg, sizeof(msg)-sizeof(long),
                                0, MSG_EXCEPT | IPC_NOWAIT)
                        NOTE:
```

```
          * System V returns LOWEST mtype first normally.
          * To simulate PRIORITY, we set:
               Highest priority = largest mtype
          * Server manually fetches messages by checking decreasing mtype.

         ALTERNATE SIMPLE METHOD:
          Receive with negative type:
             msgrcv(msgid, &msg, size, -MAX_PRIORITY, 0)
          This returns highest priority message.

      b. DISPLAY message in:
         "[Priority X] Message from client Y: TEXT"

   END LOOP



 ========================= CLIENT PROCESS =========================

 CLIENT:
   1. GET message queue:
      msgid = msgget(KEY, 0666)

   2. client_pid = getpid()

   3. LOOP FOREVER:
      a. PROMPT user:
         "Enter priority (1=low, 10=high): "
         "Enter message: "

      b. FILL msg struct:
         msg.mtype = priority      // THIS IS PRIORITY
         msg.text = input string
         msg.client_id = client_pid

      c. SEND message:
         msgsnd(msgid, &msg, sizeof(msg)-sizeof(long), 0)

   END LOOP

END PRIORITY_BASED_MESSAGE_SYSTEM
```

```c
#include <stdio.h>
#include <stdlib.h>
```

```
#include <sys/ipc.h>
#include <sys/msg.h>

#define KEY 1234
#define MAX_PRIORITY 10

struct message {
    long mtype;       // priority
    char text[200];
    int client_id;
};

int main() {
    int msgid = msgget(KEY, IPC_CREAT | 0666);
    struct message msg;

    printf("Priority Server Started...\n");

    while (1) {
        // receive highest priority message
        // negative mtype ⇒ receive message with highest priority first
        if (msgrcv(msgid, &msg, sizeof(msg) - sizeof(long),
                   -MAX_PRIORITY, 0) != -1) {

            printf("[PRIORITY %ld] From Client %d: %s",
                   msg.mtype, msg.client_id, msg.text);
        }
    }

    return 0;
}

#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <unistd.h>

#define KEY 1234

struct message {
    long mtype;       // priority
    char text[200];
    int client_id;
```

```
};

int main() {
    int msgid = msgget(KEY, 0666);
    struct message msg;
    int priority;

    msg.client_id = getpid();
    printf("Client Started (PID = %d)\n", msg.client_id);

    while (1) {
        printf("Enter priority (1=low, 10=high): ");
        scanf("%d", &priority);
        getchar();  // consume newline

        printf("Enter message: ");
        fgets(msg.text, sizeof(msg.text), stdin);

        msg.mtype = priority;

        msgsnd(msgid, &msg, sizeof(msg) - sizeof(long), 0);
    }

    return 0;
}
```

18. Shared memory calculator: One process writes two operands and an operator into shared memory. Another process reads the request, performs the calculation, and writes back the result.

```
BEGIN SHARED_MEMORY_CALCULATOR

    DEFINE key = 1234
    DEFINE STRUCT shared_data:
        int operand1
        int operand2
        char operator
        int result
        int status      // 0 = writer writes request
                        // 1 = calculator computes
                        // 2 = result ready

    CREATE shared memory: shmid = shmget(key, sizeof(shared_data), IPC_CREAT | 0666)
    ATTACH shared memory: shm_ptr = shmat(shmid)
```

```
   SET shm_ptr->status = 0


   ========================= PROCESS 1 (WRITER / CLIENT)
=========================

   LOOP FOREVER:
      WAIT UNTIL shm_ptr->status == 0

      PROMPT user: enter operand1, operand2, operator (+, -, *, /)

      WRITE shm_ptr->operand1
      WRITE shm_ptr->operand2
      WRITE shm_ptr->operator

      SET shm_ptr->status = 1       // signal calculator


      WAIT UNTIL shm_ptr->status == 2

      DISPLAY shm_ptr->result

      SET shm_ptr->status = 0       // ready for next request
   END LOOP



   ========================= PROCESS 2 (SERVER / CALCULATOR)
=========================

   LOOP FOREVER:
      WAIT UNTIL shm_ptr->status == 1

      READ operand1, operand2, operator

      SWITCH(operator):
         CASE '+': result = operand1 + operand2
         CASE '-': result = operand1 - operand2
         CASE '*': result = operand1 * operand2
         CASE '/': result = operand1 / operand2 (integer division)
      END SWITCH

      WRITE result into shm_ptr->result
```

```
        SET shm_ptr->status = 2        // result ready
    END LOOP



    ========================= CLEANUP WHEN DONE
=========================
    DETACH shared memory
    REMOVE shared memory using shmctl(shmid, IPC_RMID)

END SHARED_MEMORY_CALCULATOR
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>

struct shared_data {
    int operand1;
    int operand2;
    char operator;
    int result;
    int status;   // 0 = writer writes, 1 = calculator computes, 2 = result ready
};

int main() {
    key_t key = 1234;
    int shmid;
    struct shared_data *shm_ptr;

    // Create shared memory
    shmid = shmget(key, sizeof(struct shared_data), IPC_CREAT | 0666);
    if (shmid == -1) {
        perror("shmget");
        exit(1);
    }

    // Attach shared memory
    shm_ptr = (struct shared_data *)shmat(shmid, NULL, 0);
    if (shm_ptr == (void *)-1) {
        perror("shmat");
        exit(1);
    }
```

```
    shm_ptr->status = 0;  // ready for writer

    pid_t pid = fork();

    if (pid == 0) {
        // ========================= PROCESS 1: WRITER
=========================
        while (1) {
            while (shm_ptr->status != 0)
                ; // wait

            printf("Enter operand1: ");
            scanf("%d", &shm_ptr->operand1);

            printf("Enter operand2: ");
            scanf("%d", &shm_ptr->operand2);

            printf("Enter operator (+ - * /): ");
            scanf(" %c", &shm_ptr->operator);

            shm_ptr->status = 1; // signal calculator

            while (shm_ptr->status != 2)
                ; // wait for result

            printf("Result = %d\n", shm_ptr->result);

            shm_ptr->status = 0; // ready for next calculation
        }
    }

    else {
        // ========================= PROCESS 2: CALCULATOR
=========================
        while (1) {
            while (shm_ptr->status != 1)
                ; // wait for input

            int a = shm_ptr->operand1;
            int b = shm_ptr->operand2;
            char op = shm_ptr->operator;

            int res = 0;
```

```
        switch (op) {
            case '+': res = a + b; break;
            case '-': res = a - b; break;
            case '*': res = a * b; break;
            case '/': res = (b != 0 ? a / b : 0); break;
        }

        shm_ptr->result = res;
        shm_ptr->status = 2; // result ready
    }
  }

  // Cleanup unreachable due to infinite loops
  shmdt(shm_ptr);
  shmctl(shmid, IPC_RMID, NULL);

  return 0;
}
```

19. Multi-process shared memory sort: a. Process 1 writes an array of numbers to shared memory. b. Process 2 sorts the numbers. c. Process 3 computes the median and mean. d. Process 4 displays the final results. Synchronize using semaphores.

```
BEGIN MULTI_PROCESS_SHARED_MEMORY_SORT

  ======================== SHARED MEMORY STRUCT
===========================

  DEFINE STRUCT shared_data:
    int arr[MAX]
    int n
    float mean
    float median

  ======================== SEMAPHORES USED
===============================

  sem_t sem_input_done      // P1 → P2
  sem_t sem_sort_done       // P2 → P3
  sem_t sem_stats_done      // P3 → P4

  ======================== MAIN PROCESS (CREATE & FORK) ==================

  CREATE shared memory segment using shmget()
  ATTACH using shmat()
```

```
INITIALIZE semaphores:
   sem_init(sem_input_done, 1, 0)
   sem_init(sem_sort_done,  1, 0)
   sem_init(sem_stats_done, 1, 0)

FORK Process 1: Writer
FORK Process 2: Sorter
FORK Process 3: Stat Calculator
FORK Process 4: Display


====================== PROCESS 1 (WRITE DATA) ======================

WAIT: nothing
ACTIONS:
   Read n
   Read n numbers into shared_data.arr[]
SIGNAL sem_input_done


====================== PROCESS 2 (SORT ARRAY) ======================

WAIT sem_input_done
ACTIONS:
   Sort shared_data.arr[] (bubble sort or any)
SIGNAL sem_sort_done


====================== PROCESS 3 (MEAN + MEDIAN) ===================

WAIT sem_sort_done
ACTIONS:
   mean = sum(arr) / n
   IF n is odd:
      median = arr[n/2]
   ELSE:
      median = (arr[(n/2)-1] + arr[n/2]) / 2
WRITE results into shared memory
SIGNAL sem_stats_done


====================== PROCESS 4 (DISPLAY RESULTS) =================

WAIT sem_stats_done
```

```
    ACTIONS:
        Print sorted array
        Print mean
        Print median


    ====================== CLEANUP (parent only) ========================
    Destroy semaphores
    Detach and delete shared memory

END MULTI_PROCESS_SHARED_MEMORY_SORT

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <semaphore.h>
#include <sys/wait.h>

#define MAX 100

struct shared_data {
    int arr[MAX];
    int n;
    float mean;
    float median;
};

sem_t sem_input_done, sem_sort_done, sem_stats_done;

// ------------------------ PROCESS 2: SORT ------------------------
void sort_array(struct shared_data *shm) {
    for (int i = 0; i < shm->n - 1; i++) {
        for (int j = 0; j < shm->n - i - 1; j++) {
            if (shm->arr[j] > shm->arr[j+1]) {
                int tmp = shm->arr[j];
                shm->arr[j] = shm->arr[j+1];
                shm->arr[j+1] = tmp;
            }
        }
    }
}
```

```
// ------------------------ PROCESS 3: MEAN + MEDIAN ------------------------
void compute_stats(struct shared_data *shm) {
    int sum = 0;
    for (int i = 0; i < shm->n; i++)
        sum += shm->arr[i];

    shm->mean = (float)sum / shm->n;

    if (shm->n % 2 == 1)
        shm->median = shm->arr[shm->n/2];
    else
        shm->median = (shm->arr[(shm->n/2)-1] + shm->arr[shm->n/2]) / 2.0;
}

int main() {
    key_t key = 1234;
    int shmid;

    shmid = shmget(key, sizeof(struct shared_data), IPC_CREAT | 0666);
    struct shared_data *shm = shmat(shmid, NULL, 0);

    // Initialize semaphores
    sem_init(&sem_input_done, 1, 0);
    sem_init(&sem_sort_done,  1, 0);
    sem_init(&sem_stats_done, 1, 0);

    // ------------------------ PROCESS 1: INPUT ------------------------
    if (fork() == 0) {
        printf("Enter number of elements: ");
        scanf("%d", &shm->n);

        printf("Enter %d numbers:\n", shm->n);
        for (int i = 0; i < shm->n; i++)
            scanf("%d", &shm->arr[i]);

        sem_post(&sem_input_done);
        exit(0);
    }

    // ------------------------ PROCESS 2: SORT ------------------------
    if (fork() == 0) {
        sem_wait(&sem_input_done);
        sort_array(shm);
        sem_post(&sem_sort_done);
```

```
        exit(0);
    }

    // ------------------------ PROCESS 3: COMPUTE MEAN + MEDIAN ------------------------
    if (fork() == 0) {
        sem_wait(&sem_sort_done);
        compute_stats(shm);
        sem_post(&sem_stats_done);
        exit(0);
    }

    // ------------------------ PROCESS 4: DISPLAY RESULTS ------------------------
    if (fork() == 0) {
        sem_wait(&sem_stats_done);

        printf("\nSorted Array: ");
        for (int i = 0; i < shm->n; i++)
            printf("%d ", shm->arr[i]);

        printf("\nMean: %.2f\n", shm->mean);
        printf("Median: %.2f\n", shm->median);

        exit(0);
    }

    // Parent waits for all children
    wait(NULL);
    wait(NULL);
    wait(NULL);
    wait(NULL);

    // Cleanup
    sem_destroy(&sem_input_done);
    sem_destroy(&sem_sort_done);
    sem_destroy(&sem_stats_done);
    shmdt(shm);
    shmctl(shmid, IPC_RMID, NULL);

    return 0;
}
```

20. Shared memory file transfer: Implement a program where one process reads data from a file and writes into shared memory in chunks, and another process reconstructs the file.
BEGIN SHARED_MEMORY_FILE_TRANSFER

```
==================== SHARED MEMORY STRUCTURE =====================
DEFINE CHUNK_SIZE = 256
DEFINE STRUCT shared_data:
    char buffer[CHUNK_SIZE]
    int  bytes_read         // number of bytes written in buffer
    int  done               // flag: 0 = more data, 1 = last chunk

==================== SEMAPHORES USED ==============================
sem_t sem_empty    // writer allowed to write
sem_t sem_full     // reader allowed to read

==================== MAIN PROGRAM SETUP ===========================

1. CREATE shared memory using shmget()
2. ATTACH using shmat()
3. INITIALIZE sem_empty = 1   // writer starts first
4. INITIALIZE sem_full  = 0   // reader waits
5. FORK writer process
6. FORK reader process


==================== PROCESS 1 (WRITER: READ FILE → SHM) =========

P1:
    OPEN input file for reading

    LOOP:
        WAIT sem_empty            // ensure buffer is free to write

        READ up to CHUNK_SIZE bytes from file into shm_ptr->buffer
        STORE number of bytes in shm_ptr->bytes_read

        IF bytes_read < CHUNK_SIZE THEN
            shm_ptr->done = 1      // last chunk
        ELSE
            shm_ptr->done = 0      // more data

        SIGNAL sem_full           // allow reader to consume

        IF done == 1 THEN BREAK

    CLOSE file
    EXIT
```

```
==================== PROCESS 2 (READER: SHM → OUTPUT FILE) ======

P2:
  OPEN output file for writing

  LOOP:
    WAIT sem_full          // wait for buffer with new data

    WRITE shm_ptr->buffer[0 .. bytes_read-1] into output file

    IF shm_ptr->done == 1 THEN BREAK

    SIGNAL sem_empty        // tell writer to write next chunk

  SIGNAL sem_empty  // final signal in case writer needs it
  CLOSE file
  EXIT


==================== CLEANUP IN MAIN PROCESS ====================
WAIT for both children
DESTROY semaphores
DETACH shared memory
DELETE shared memory segment using shmctl()

END SHARED_MEMORY_FILE_TRANSFER
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <string.h>

#define CHUNK_SIZE 256

struct shared_data {
  char buffer[CHUNK_SIZE];
  int bytes_read;
```

```
    int done;
};

sem_t sem_empty, sem_full;

int main() {
    key_t key = 1234;
    int shmid;

    shmid = shmget(key, sizeof(struct shared_data), IPC_CREAT | 0666);
    struct shared_data *shm = (struct shared_data *)shmat(shmid, NULL, 0);

    // Initialize semaphores
    sem_init(&sem_empty, 1, 1);
    sem_init(&sem_full,  1, 0);

    pid_t writer = fork();

    if (writer == 0) {
        // ===================== WRITER PROCESS =====================
        FILE *in = fopen("input.txt", "rb");
        if (!in) { perror("file open"); exit(1); }

        while (1) {
            sem_wait(&sem_empty);

            shm->bytes_read = fread(shm->buffer, 1, CHUNK_SIZE, in);

            if (shm->bytes_read < CHUNK_SIZE)
                shm->done = 1;     // last chunk
            else
                shm->done = 0;

            sem_post(&sem_full);

            if (shm->done == 1)
                break;
        }

        fclose(in);
        exit(0);
    }

    pid_t reader = fork();
```

```
    if (reader == 0) {
        // ====================== READER PROCESS ======================
        FILE *out = fopen("output.txt", "wb");
        if (!out) { perror("file open"); exit(1); }

        while (1) {
            sem_wait(&sem_full);

            fwrite(shm->buffer, 1, shm->bytes_read, out);

            if (shm->done == 1)
                break;

            sem_post(&sem_empty);
        }

        sem_post(&sem_empty);
        fclose(out);
        exit(0);
    }

    // Parent waits
    wait(NULL);
    wait(NULL);

    // Cleanup
    sem_destroy(&sem_empty);
    sem_destroy(&sem_full);
    shmdt(shm);
    shmctl(shmid, IPC_RMID, NULL);

    return 0;
}
```

21. Dining philosophers problem: Implement the classical dining philosophers synchronization problem using semaphores.

```
BEGIN DINING_PHILOSOPHERS

    CONSTANT N = 5   // number of philosophers and forks

    CREATE array sem_fork[N]   // one semaphore per fork
    FOR each i in 0..N-1:
         sem_init(sem_fork[i], 1, 1)   // each fork initially available
```

```
    CREATE array of processes via fork() or threads

    FOR philosopher i:
        LOOP forever:

            THINK for some time

            IF i is even THEN
                WAIT sem_fork[i]           // pick left fork
                WAIT sem_fork[(i+1) % N]   // pick right fork
            ELSE
                WAIT sem_fork[(i+1) % N]   // pick right fork first
                WAIT sem_fork[i]           // pick left fork
            END IF

            EAT for some time

            SIGNAL sem_fork[i]             // release left fork
            SIGNAL sem_fork[(i+1) % N]     // release right fork

        END LOOP

    PARENT waits for all philosophers (optional)

    FOR all semaphores:
        sem_destroy()

END DINING_PHILOSOPHERS
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

#define N 5    // number of philosophers

sem_t forks[N];

void *philosopher(void *num) {
    int id = *(int *)num;

    while (1) {
```

```c
        printf("Philosopher %d is thinking...\n", id);
        sleep(1);

        if (id % 2 == 0) {
            sem_wait(&forks[id]);                // pick left fork
            sem_wait(&forks[(id + 1) % N]);      // pick right fork
        } else {
            sem_wait(&forks[(id + 1) % N]);      // pick right fork
            sem_wait(&forks[id]);                // pick left fork
        }

        printf("Philosopher %d is eating...\n", id);
        sleep(1);

        sem_post(&forks[id]);                    // put down left fork
        sem_post(&forks[(id + 1) % N]);          // put down right fork
    }
}

int main() {
    pthread_t threads[N];
    int ids[N];

    // Initialize semaphores
    for (int i = 0; i < N; i++) {
        sem_init(&forks[i], 0, 1);
    }

    // Create philosopher threads
    for (int i = 0; i < N; i++) {
        ids[i] = i;
        pthread_create(&threads[i], NULL, philosopher, &ids[i]);
    }

    // Join (never ends)
    for (int i = 0; i < N; i++) {
        pthread_join(threads[i], NULL);
    }

    // Destroy semaphores (never reached in infinite loop)
    for (int i = 0; i < N; i++) {
        sem_destroy(&forks[i]);
    }
```

```
    return 0;
}
```

22. Producer–Consumer with bounded buffer: Use shared memory and semaphores to implement multiple producers and multiple consumers.

```
BEGIN MULTI_PRODUCER_CONSUMER_SYSTEM

    ==================== SHARED MEMORY STRUCT =====================
    DEFINE BUFFER_SIZE = N

    STRUCT shared_data:
        int buffer[BUFFER_SIZE]
        int in      // producer index
        int out     // consumer index

    ===================== SEMAPHORES USED ==========================
    sem_t mutex       // binary semaphore for mutual exclusion
    sem_t empty       // counts empty slots  (initial = BUFFER_SIZE)
    sem_t full        // counts filled slots (initial = 0)

    ===================== MAIN PROCESS SETUP ======================

    1. CREATE shared memory (shmget)
    2. ATTACH to shared memory (shmat)
    3. INITIALIZE:
           shm_ptr->in = 0
           shm_ptr->out = 0

    4. INIT semaphores:
           sem_init(mutex, 1, 1)
           sem_init(empty, 1, BUFFER_SIZE)
           sem_init(full,  1, 0)

    5. FORK P producer processes
    6. FORK C consumer processes


    ===================== PRODUCER PROCESS LOGIC ===================

    LOOP FOREVER:
        PRODUCER generates item

        WAIT(empty)     // ensure buffer has space
        WAIT(mutex)     // enter critical section
```

```
        buffer[in] = item
        in = (in + 1) mod BUFFER_SIZE

      POST(mutex)     // leave critical section
      POST(full)      // signal a filled slot

      sleep(optional) // slow producer
   END LOOP


   ==================== CONSUMER PROCESS LOGIC ===================

   LOOP FOREVER:
      WAIT(full)      // ensure data is available
      WAIT(mutex)     // enter critical section

        item = buffer[out]
        out = (out + 1) mod BUFFER_SIZE

      POST(mutex)     // leave critical section
      POST(empty)     // signal an empty slot

      PRINT consumed item
      sleep(optional)
   END LOOP


   ==================== CLEANUP IN PARENT =======================
   WAIT for all children
   sem_destroy(mutex)
   sem_destroy(empty)
   sem_destroy(full)
   shmdt & shmctl()

END MULTI_PRODUCER_CONSUMER_SYSTEM
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>
#include <semaphore.h>
```

```
#define BUFFER_SIZE 5
#define PRODUCERS 2
#define CONSUMERS 2

struct shared_data {
    int buffer[BUFFER_SIZE];
    int in, out;
};

sem_t mutex, empty, full;

// ---------------------- PRODUCER ----------------------
void producer(struct shared_data *shm, int id) {
    int item = 1;

    while (1) {
        item = rand() % 100;   // generate data

        sem_wait(&empty);
        sem_wait(&mutex);

        shm->buffer[shm->in] = item;
        printf("Producer %d produced: %d at index %d\n", id, item, shm->in);
        shm->in = (shm->in + 1) % BUFFER_SIZE;

        sem_post(&mutex);
        sem_post(&full);

        sleep(1);
    }
}

// ---------------------- CONSUMER ----------------------
void consumer(struct shared_data *shm, int id) {
    int item;

    while (1) {
        sem_wait(&full);
        sem_wait(&mutex);

        item = shm->buffer[shm->out];
        printf("Consumer %d consumed: %d from index %d\n", id, item, shm->out);
        shm->out = (shm->out + 1) % BUFFER_SIZE;
```

```
        sem_post(&mutex);
        sem_post(&empty);

        sleep(1);
    }
}

int main() {
    key_t key = 1234;
    int shmid;
    struct shared_data *shm;

    // Shared memory creation
    shmid = shmget(key, sizeof(struct shared_data), IPC_CREAT | 0666);
    shm = (struct shared_data *)shmat(shmid, NULL, 0);

    shm->in = 0;
    shm->out = 0;

    // Initialize semaphores
    sem_init(&mutex, 1, 1);
    sem_init(&empty, 1, BUFFER_SIZE);
    sem_init(&full,  1, 0);

    // Fork producers
    for (int i = 0; i < PRODUCERS; i++) {
        if (fork() == 0) {
            producer(shm, i + 1);
            exit(0);
        }
    }

    // Fork consumers
    for (int i = 0; i < CONSUMERS; i++) {
        if (fork() == 0) {
            consumer(shm, i + 1);
            exit(0);
        }
    }

    // Parent waits
    for (int i = 0; i < PRODUCERS + CONSUMERS; i++)
        wait(NULL);
```

```
    // Cleanup
    sem_destroy(&mutex);
    sem_destroy(&empty);
    sem_destroy(&full);
    shmdt(shm);
    shmctl(shmid, IPC_RMID, NULL);

    return 0;
}
```

23. Reader–Writer with priority: Implement the reader–writer problem using semaphores such that writers are given higher priority than readers.

```
BEGIN WRITER_PRIORITY_READER_WRITER

    ===================== SHARED VARIABLES =====================
    int read_count = 0          // number of active readers
    int write_count = 0         // number of writers waiting or active

    SEMAPHORES:
        sem_t mutex_readcount       // protects read_count
        sem_t mutex_writecount      // protects write_count
        sem_t rw_mutex              // controls access to shared resource
        sem_t queue_mutex           // blocks readers if writers waiting


    ===================== READER LOGIC (Writer Priority) =====================

    READER:
        WAIT(queue_mutex)               // check if writer waiting
        WAIT(mutex_readcount)
            read_count++
            IF read_count == 1:
                WAIT(rw_mutex)          // lock shared resource (1st reader)
        SIGNAL(mutex_readcount)
        SIGNAL(queue_mutex)

        ---- CRITICAL SECTION: READ -----

        WAIT(mutex_readcount)
            read_count--
            IF read_count == 0:
                SIGNAL(rw_mutex)        // last reader frees resource
        SIGNAL(mutex_readcount)
```

```
    ==================== WRITER LOGIC (Priority Enforcement)
====================

    WRITER:
      WAIT(mutex_writecount)
        write_count++
        IF write_count == 1:
            WAIT(queue_mutex)       // block new readers
      SIGNAL(mutex_writecount)

      WAIT(rw_mutex)                // writer gets exclusive access

      ---- CRITICAL SECTION: WRITE ----

      SIGNAL(rw_mutex)

      WAIT(mutex_writecount)
        write_count--
        IF write_count == 0:
            SIGNAL(queue_mutex)     // allow readers again
      SIGNAL(mutex_writecount)



    ==================== MAIN PROCESS SETUP ====================

    INIT semaphores:
      mutex_readcount = 1
      mutex_writecount = 1
      rw_mutex = 1
      queue_mutex = 1

    CREATE multiple reader threads
    CREATE multiple writer threads

END WRITER_PRIORITY_READER_WRITER
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
```

```
#include <unistd.h>

sem_t mutex_readcount, mutex_writecount, rw_mutex, queue_mutex;
int read_count = 0, write_count = 0;

void *reader(void *arg) {
    int id = *(int *)arg;

    while (1) {
        // Entry section (writer priority)
        sem_wait(&queue_mutex);
        sem_wait(&mutex_readcount);

        read_count++;
        if (read_count == 1)
            sem_wait(&rw_mutex); // first reader locks data

        sem_post(&mutex_readcount);
        sem_post(&queue_mutex);

        // ------------ Critical Section ------------
        printf("Reader %d is reading...\n", id);
        sleep(1);

        // Exit section
        sem_wait(&mutex_readcount);
        read_count--;
        if (read_count == 0)
            sem_post(&rw_mutex); // last reader unlocks data
        sem_post(&mutex_readcount);

        sleep(1);
    }
}

void *writer(void *arg) {
    int id = *(int *)arg;

    while (1) {
        // Entry section (High Priority)
        sem_wait(&mutex_writecount);

        write_count++;
        if (write_count == 1)
```

```c
            sem_wait(&queue_mutex); // block readers

        sem_post(&mutex_writecount);

        sem_wait(&rw_mutex); // writer exclusive access

        // ------------ Critical Section ------------
        printf("Writer %d is writing...\n", id);
        sleep(1);

        sem_post(&rw_mutex);

        // Exit
        sem_wait(&mutex_writecount);
        write_count--;
        if (write_count == 0)
            sem_post(&queue_mutex); // let readers in again
        sem_post(&mutex_writecount);

        sleep(1);
    }
}

int main() {
    pthread_t r[3], w[2];
    int r_id[3] = {1, 2, 3};
    int w_id[2] = {1, 2};

    sem_init(&mutex_readcount, 0, 1);
    sem_init(&mutex_writecount, 0, 1);
    sem_init(&rw_mutex,        0, 1);
    sem_init(&queue_mutex,     0, 1);

    // Create threads
    for (int i = 0; i < 3; i++)
        pthread_create(&r[i], NULL, reader, &r_id[i]);

    for (int i = 0; i < 2; i++)
        pthread_create(&w[i], NULL, writer, &w_id[i]);

    // Join (never ends)
    for (int i = 0; i < 3; i++) pthread_join(r[i], NULL);
    for (int i = 0; i < 2; i++) pthread_join(w[i], NULL);
```

```
    return 0;
}
```

24. Concurrent TCP echo server: Implement an echo server using TCP sockets that handles multiple clients concurrently using fork() or threads.BEGIN CONCURRENT_TCP_ECHO_SERVER

```
    ========================== COMMON PARAMETERS
==========================
    PORT = 8080
    BUFFER_SIZE = 1024

    ========================== SERVER SIDE
==================================

    SERVER:
        1. CREATE listening socket:
             server_fd = socket(AF_INET, SOCK_STREAM, 0)

        2. PREPARE server_addr:
             sin_family      = AF_INET
             sin_addr.s_addr = INADDR_ANY
             sin_port        = htons(PORT)

        3. BIND server_fd to server_addr
        4. LISTEN(server_fd, backlog=10)

        LOOP forever:
            5. ACCEPT incoming client:
                 new_socket = accept(server_fd)

            6. FORK a child process:
                 IF child:
                     a. CLOSE server_fd (child does not accept)
                     b. LOOP:
                            RECEIVE message from client
                            SEND same message back (echo)
                        END LOOP when client disconnects
                     c. CLOSE new_socket
                     d. EXIT child
                 ELSE (parent):
                     a. CLOSE new_socket (parent doesn't use it)
        END LOOP
```

```
    ========================= CLIENT SIDE
=================================

    CLIENT:
        1. CREATE TCP socket: client_fd
        2. PREPARE server_addr:
              server IP = "127.0.0.1"
              server port = PORT

        3. CONNECT(client_fd, server_addr)

        LOOP forever:
            a. READ user input
            b. SEND message to server
            c. RECEIVE echoed message
            d. PRINT echoed message
        END LOOP

END CONCURRENT_TCP_ECHO_SERVER

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/wait.h>

#define PORT 8080
#define BUFFER 1024

int main() {
    int server_fd, new_socket;
    char buffer[BUFFER];
    struct sockaddr_in address;
    socklen_t addrlen = sizeof(address);

    server_fd = socket(AF_INET, SOCK_STREAM, 0);

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);
```

```
    bind(server_fd, (struct sockaddr*)&address, sizeof(address));
    listen(server_fd, 10);

    printf("Concurrent TCP Echo Server running...\n");

    while (1) {
        new_socket = accept(server_fd, (struct sockaddr*)&address, &addrlen);
        printf("Client connected.\n");

        if (fork() == 0) {
            close(server_fd);  // child does not accept more

            while (1) {
                int n = recv(new_socket, buffer, BUFFER, 0);
                if (n <= 0) break;
                buffer[n] = '\0';

                send(new_socket, buffer, n, 0);
            }

            printf("Client disconnected (child process).\n");
            close(new_socket);
            exit(0);
        }

        close(new_socket);  // parent does not use client socket
    }

    close(server_fd);
    return 0;
}

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 8080
#define BUFFER 1024

int main() {
    int client_fd;
    char buffer[BUFFER];
    struct sockaddr_in server_addr;
```

```
    client_fd = socket(AF_INET, SOCK_STREAM, 0);

    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");

    connect(client_fd, (struct sockaddr*)&server_addr, sizeof(server_addr));

    while (1) {
        printf("Enter message: ");
        fgets(buffer, BUFFER, stdin);

        send(client_fd, buffer, strlen(buffer), 0);

        int n = recv(client_fd, buffer, BUFFER, 0);
        buffer[n] = '\0';
        printf("Echo: %s\n", buffer);
    }

    close(client_fd);
    return 0;
}
```

25. UDP file transfer: Implement a client-server application where the client requests a file and the server sends it via UDP in chunks, with acknowledgments.

```
BEGIN UDP_FILE_TRANSFER_SYSTEM

    CONSTANT CHUNK_SIZE = 512 bytes
    CONSTANT PORT = 9090

    STRUCT packet:
        int seq_no             // sequence number
        int bytes              // bytes of valid data in chunk
        char data[CHUNK_SIZE]  // file contents

    STRUCT ack:
        int seq_no             // acknowledgment number


    ================================ SERVER LOGIC
================================

    SERVER:
```

1. CREATE UDP socket → server_fd = socket(AF_INET, SOCK_DGRAM, 0)

2. BIND server_fd to PORT

3. LOOP FOREVER:
    a. WAIT for file request from client using recvfrom()
    b. OPEN requested file in binary mode: fopen()

    c. seq_no = 0

    d. LOOP:
        i. READ CHUNK_SIZE bytes from file into pkt.data
        ii. SET pkt.bytes = size read
        iii. SET pkt.seq_no = seq_no

        iv. SEND packet to client via sendto()

        v. WAIT for ACK from client (recvfrom())
           If ack.seq_no != seq_no → resend packet

        vi. seq_no++

        vii. IF pkt.bytes < CHUNK_SIZE → LAST PACKET; break
      END LOOP

    e. CLOSE file

END LOOP

================================ CLIENT LOGIC
================================

CLIENT:
1. CREATE UDP socket → client_fd

2. SETUP server address (IP + PORT)

3. INPUT filename from user

4. SEND filename to server using sendto()

5. OPEN an output file locally for writing

6. expected_seq_no = 0

7. LOOP FOREVER:
    a. RECEIVE packet pkt from server (recvfrom())

    b. IF pkt.seq_no == expected_seq_no:
        WRITE pkt.data[pkt.bytes] to output file
        expected_seq_no++
        PREPARE ack.seq_no = pkt.seq_no
        SEND ack to server
      ELSE:
        SEND ack with last acknowledged seq_no (retransmit request)

    c. IF pkt.bytes < CHUNK_SIZE → break   // end of file

8. CLOSE output file
9. EXIT PROGRAM

END UDP_FILE_TRANSFER_SYSTEM

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 9090
#define CHUNK_SIZE 512

struct packet {
    int seq_no;
    int bytes;
    char data[CHUNK_SIZE];
};

struct ack {
    int seq_no;
};

int main() {
    int server_fd;
    struct sockaddr_in server_addr, client_addr;
    socklen_t addrlen = sizeof(client_addr);
```

```
server_fd = socket(AF_INET, SOCK_DGRAM, 0);

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

bind(server_fd, (struct sockaddr*)&server_addr, sizeof(server_addr));

printf("UDP File Server running on port %d...\n", PORT);

char filename[100];
struct packet pkt;
struct ack ack_pkt;

while (1) {
    // Receive filename
    recvfrom(server_fd, filename, sizeof(filename), 0,
             (struct sockaddr*)&client_addr, &addrlen);

    printf("Client requested file: %s\n", filename);

    FILE *fp = fopen(filename, "rb");
    if (!fp) {
        perror("File open");
        continue;
    }

    int seq_no = 0;

    while (1) {
        pkt.seq_no = seq_no;
        pkt.bytes = fread(pkt.data, 1, CHUNK_SIZE, fp);

        // Send file chunk
        sendto(server_fd, &pkt, sizeof(pkt), 0,
               (struct sockaddr*)&client_addr, addrlen);

        // Wait for ACK
        recvfrom(server_fd, &ack_pkt, sizeof(ack_pkt), 0,
                 (struct sockaddr*)&client_addr, &addrlen);

        if (ack_pkt.seq_no != seq_no) {
            // resend same packet
```

```c
            continue;
        }

        seq_no++;

        if (pkt.bytes < CHUNK_SIZE)
            break;  // last packet
    }

    fclose(fp);
    printf("File transfer completed.\n");
}

close(server_fd);
return 0;
}

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 9090
#define CHUNK_SIZE 512

struct packet {
    int seq_no;
    int bytes;
    char data[CHUNK_SIZE];
};

struct ack {
    int seq_no;
};

int main() {
    int client_fd;
    struct sockaddr_in server_addr;
    socklen_t addrlen = sizeof(server_addr);

    client_fd = socket(AF_INET, SOCK_DGRAM, 0);

    server_addr.sin_family = AF_INET;
```

```
server_addr.sin_port = htons(PORT);
server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");

char filename[100];
printf("Enter filename to download: ");
scanf("%s", filename);

// Send file request
sendto(client_fd, filename, strlen(filename)+1, 0,
       (struct sockaddr*)&server_addr, addrlen);

FILE *fp = fopen("downloaded_file", "wb");
if (!fp) {
    perror("open");
    exit(1);
}

struct packet pkt;
struct ack ack_pkt;

int expected_seq = 0;

while (1) {
    recvfrom(client_fd, &pkt, sizeof(pkt), 0,
             (struct sockaddr*)&server_addr, &addrlen);

    if (pkt.seq_no == expected_seq) {
        fwrite(pkt.data, 1, pkt.bytes, fp);

        ack_pkt.seq_no = pkt.seq_no;
        sendto(client_fd, &ack_pkt, sizeof(ack_pkt), 0,
               (struct sockaddr*)&server_addr, addrlen);

        expected_seq++;

        if (pkt.bytes < CHUNK_SIZE)
            break;  // end of file
    } else {
        // resend last ACK to request retransmission
        ack_pkt.seq_no = expected_seq - 1;
        sendto(client_fd, &ack_pkt, sizeof(ack_pkt), 0,
               (struct sockaddr*)&server_addr, addrlen);
    }
}
```

```
    fclose(fp);
    printf("File downloaded successfully as 'downloaded_file'\n");

    close(client_fd);
    return 0;
}
```

26. Distributed computation server: Create a TCP server that accepts computation requests (factorial, matrix multiplication, string reversal) from multiple clients and returns results.

```
BEGIN DISTRIBUTED_COMPUTATION_SERVER

    ========================== COMMON DEFINITIONS
===========================
    PORT = 8080
    BUFFER_SIZE = 1024

    REQUEST FORMAT SENT BY CLIENT:
        option:
            1 → factorial
            2 → 2x2 matrix multiplication
            3 → string reversal

        For factorial:
            send: "1 number"

        For matrix multiplication:
            send: "2 a11 a12 a21 a22 b11 b12 b21 b22"

        For string reversal:
            send: "3 some_string"


    ========================== SERVER LOGIC
=================================

    SERVER:
        1. CREATE TCP socket (server_fd)
        2. CONFIGURE sockaddr_in (AF_INET, INADDR_ANY, PORT)
        3. BIND socket
        4. LISTEN(server_fd, backlog=10)

        LOOP FOREVER:
            5. ACCEPT client → new_socket
```

```
      6. FORK a new process:
         IF child:
            a. CLOSE server_fd
            b. LOOP:
                  i.   recv() request
                  ii.  parse option
                  iii. PERFORM computation:
                           CASE 1: factorial(n)
                           CASE 2: multiply matrices A and B
                           CASE 3: reverse string
                  iv.  send() result back
               END LOOP when client disconnects
            c. CLOSE new_socket
            d. EXIT child
         ELSE:
            CLOSE new_socket (parent does not handle client)
   END LOOP


  ========================== CLIENT LOGIC
==================================

  CLIENT:
     1. CREATE TCP socket
     2. CONNECT to server (127.0.0.1:PORT)

     LOOP:
        a. DISPLAY MENU:
              1 factorial
              2 matrix multiplication
              3 string reversal

        b. READ user choice
        c. PREPARE request string (as specified above)
        d. SEND request over socket
        e. RECEIVE response
        f. DISPLAY result
     END LOOP

END DISTRIBUTED_COMPUTATION_SERVER

#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 8080
#define BUFFER 1024

long factorial(int n) {
    long f = 1;
    for (int i = 1; i <= n; i++) f *= i;
    return f;
}

void reverse_string(char *str) {
    int l = 0, r = strlen(str) - 1;
    while (l < r) {
        char t = str[l];
        str[l] = str[r];
        str[r] = t;
        l++; r--;
    }
}

void multiply_2x2(int *A, int *B, int *C) {
    C[0] = A[0]*B[0] + A[1]*B[2];
    C[1] = A[0]*B[1] + A[1]*B[3];
    C[2] = A[2]*B[0] + A[3]*B[2];
    C[3] = A[2]*B[1] + A[3]*B[3];
}

int main() {
    int server_fd, new_socket;
    char buffer[BUFFER];
    struct sockaddr_in address;
    socklen_t addrlen = sizeof(address);

    server_fd = socket(AF_INET, SOCK_STREAM, 0);

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    bind(server_fd, (struct sockaddr*)&address, sizeof(address));
    listen(server_fd, 10);
```

```
printf("Distributed Computation Server running...\n");

while (1) {
    new_socket = accept(server_fd, (struct sockaddr*)&address, &addrlen);
    printf("Client connected.\n");

    if (fork() == 0) {
        close(server_fd);

        while (1) {
            int n = recv(new_socket, buffer, BUFFER, 0);
            if (n <= 0) break;
            buffer[n] = '\0';

            int option;
            sscanf(buffer, "%d", &option);

            char response[BUFFER];

            if (option == 1) {
                int num;
                sscanf(buffer, "%d %d", &option, &num);
                long f = factorial(num);
                sprintf(response, "Factorial = %ld", f);
            }

            else if (option == 2) {
                int A[4], B[4], C[4];
                sscanf(buffer,
                       "%d %d %d %d %d %d %d %d %d",
                       &option,
                       &A[0], &A[1], &A[2], &A[3],
                       &B[0], &B[1], &B[2], &B[3]);
                multiply_2x2(A, B, C);
                sprintf(response, "Matrix Result:\n%d %d\n%d %d",
                        C[0], C[1], C[2], C[3]);
            }

            else if (option == 3) {
                char msg[BUFFER];
                strcpy(msg, buffer + 2);
                reverse_string(msg);
                sprintf(response, "Reversed: %s", msg);
```

```c
            }

            send(new_socket, response, strlen(response), 0);
        }

        printf("Client disconnected.\n");
        close(new_socket);
        exit(0);
    }

    close(new_socket);
  }

  return 0;
}

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 8080
#define BUFFER 1024

int main() {
  int client_fd;
  char buffer[BUFFER];
  struct sockaddr_in server_addr;

  client_fd = socket(AF_INET, SOCK_STREAM, 0);

  server_addr.sin_family = AF_INET;
  server_addr.sin_port = htons(PORT);
  server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");

  connect(client_fd, (struct sockaddr*)&server_addr, sizeof(server_addr));

  while (1) {
    printf("\n--- Menu ---\n");
    printf("1. Factorial\n");
    printf("2. Matrix Multiplication (2x2)\n");
    printf("3. String Reversal\n");
    printf("Enter option: ");
```

```
        int option;
        scanf("%d", &option);

        char request[BUFFER];

        if (option == 1) {
            int n;
            printf("Enter number: ");
            scanf("%d", &n);
            sprintf(request, "1 %d", n);
        }

        else if (option == 2) {
            int A[4], B[4];
            printf("Enter A matrix (a11 a12 a21 a22): ");
            scanf("%d %d %d %d", &A[0], &A[1], &A[2], &A[3]);
            printf("Enter B matrix (b11 b12 b21 b22): ");
            scanf("%d %d %d %d", &B[0], &B[1], &B[2], &B[3]);

            sprintf(request, "2 %d %d %d %d %d %d %d %d",
                    A[0], A[1], A[2], A[3],
                    B[0], B[1], B[2], B[3]);
        }

        else if (option == 3) {
            char msg[BUFFER];
            printf("Enter string: ");
            scanf(" %[^\n]s", msg);
            sprintf(request, "3 %s", msg);
        }

        send(client_fd, request, strlen(request), 0);

        int n = recv(client_fd, buffer, BUFFER, 0);
        buffer[n] = '\0';
        printf("\nResult:\n%s\n", buffer);
    }

    close(client_fd);
    return 0;
}
```

27. Multi-user chat server: Implement a group chat system using TCP sockets where multiple clients can join, send messages, and receive broadcasts from the server.

```
BEGIN MULTI_USER_CHAT_SYSTEM

============================== SERVER SIDE ==============================

SERVER:
   1. CREATE TCP socket: server_fd = socket(AF_INET, SOCK_STREAM, 0)
   2. INITIALIZE sockaddr_in (PORT = 8080)
   3. BIND server_fd to PORT
   4. LISTEN(server_fd, 10)

   5. CREATE global array client_sockets[]
   6. CREATE a mutex lock for modifying client list

   LOOP FOREVER:
      a. ACCEPT new client → new_socket
      b. ADD new_socket to client_sockets[] (inside mutex)
      c. CREATE a new THREAD for this client:
           THREAD runs handle_client(new_socket)

   END LOOP


FUNCTION handle_client(socket):
   LOOP FOREVER:
      1. RECEIVE message from this client
      2. IF client disconnected → REMOVE from list → EXIT THREAD
      3. BROADCAST message to ALL connected clients (except sender)
        (Use mutex while iterating client list)
   END LOOP



============================== CLIENT SIDE ==============================

CLIENT:
   1. CREATE socket
   2. CONNECT to server (127.0.0.1:8080)

   3. CREATE THREAD receiver_thread:
        LOOP forever:
           recv() message and print it

   4. MAIN LOOP:
        read line from user
```

send to server

5. On exit, close socket

END MULTI_USER_CHAT_SYSTEM

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <pthread.h>

#define PORT 8080
#define MAX_CLIENTS 100
#define BUFFER 1024

int clients[MAX_CLIENTS];
int client_count = 0;
pthread_mutex_t lock;

void broadcast_message(char *msg, int sender_socket) {
    pthread_mutex_lock(&lock);
    for (int i = 0; i < client_count; i++) {
        if (clients[i] != sender_socket) {
            send(clients[i], msg, strlen(msg), 0);
        }
    }
    pthread_mutex_unlock(&lock);
}

void *handle_client(void *arg) {
    int sock = *(int *)arg;
    char buffer[BUFFER];

    while (1) {
        int n = recv(sock, buffer, BUFFER, 0);
        if (n <= 0) break;

        buffer[n] = '\0';
        broadcast_message(buffer, sock);
    }

    // Client disconnected
```

```
    pthread_mutex_lock(&lock);
    for (int i = 0; i < client_count; i++) {
        if (clients[i] == sock) {
            clients[i] = clients[client_count - 1];
            break;
        }
    }
    client_count--;
    pthread_mutex_unlock(&lock);

    close(sock);
    pthread_exit(NULL);
}

int main() {
    int server_fd, new_socket;
    struct sockaddr_in server_addr, client_addr;
    socklen_t addrlen = sizeof(client_addr);

    pthread_mutex_init(&lock, NULL);

    server_fd = socket(AF_INET, SOCK_STREAM, 0);

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

    bind(server_fd, (struct sockaddr*)&server_addr, sizeof(server_addr));
    listen(server_fd, 10);

    printf("Chat Server running on port %d...\n", PORT);

    while (1) {
        new_socket = accept(server_fd, (struct sockaddr*)&client_addr, &addrlen);

        pthread_mutex_lock(&lock);
        clients[client_count++] = new_socket;
        pthread_mutex_unlock(&lock);

        pthread_t t;
        pthread_create(&t, NULL, handle_client, &new_socket);
        pthread_detach(t);

        printf("Client connected.\n");
```

```
    }

    close(server_fd);
    return 0;
}

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <arpa/inet.h>

#define PORT 8080
#define BUFFER 1024

int server_socket;

void *receive_messages(void *arg) {
    char buffer[BUFFER];
    while (1) {
        int n = recv(server_socket, buffer, BUFFER, 0);
        if (n > 0) {
            buffer[n] = '\0';
            printf("%s", buffer);
        }
    }
}

int main() {
    struct sockaddr_in server_addr;
    char msg[BUFFER];

    server_socket = socket(AF_INET, SOCK_STREAM, 0);

    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");

    connect(server_socket, (struct sockaddr*)&server_addr, sizeof(server_addr));

    pthread_t recv_thread;
    pthread_create(&recv_thread, NULL, receive_messages, NULL);
    pthread_detach(recv_thread);
```

```
    while (1) {
        fgets(msg, BUFFER, stdin);
        send(server_socket, msg, strlen(msg), 0);
    }

    close(server_socket);
    return 0;
}
```

28. Implement a program that uses the fork() system call to create five child processes and assigns a distinct operation to each child.

```
BEGIN MULTI_CHILD_PROCESS_PROGRAM

    DEFINE NUM_CHILDREN = 5
    DECLARE pid variable

    FOR i FROM 1 TO 5 DO
        pid = fork()

        IF pid < 0 THEN
            ERROR (fork failed)

        ELSE IF pid == 0 THEN     // CHILD PROCESS

            SWITCH(i):

                CASE 1:
                    perform operation A (e.g., print Fibonacci)
                    EXIT child

                CASE 2:
                    perform operation B (e.g., compute factorial)
                    EXIT child

                CASE 3:
                    perform operation C (e.g., print even numbers)
                    EXIT child

                CASE 4:
                    perform operation D (e.g., print odd numbers)
                    EXIT child

                CASE 5:
                    perform operation E (e.g., print current PID)
```

```
                EXIT child

            END SWITCH

        END IF
    END FOR

    // PARENT PROCESS
    LOOP 5 times:
        wait() for each child to finish

END PROGRAM
```

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

// Simple functions for demonstration

void fibonacci() {
    int a = 0, b = 1, c;
    printf("Child 1 (Fibonacci): ");
    for (int i = 0; i < 10; i++) {
        printf("%d ", a);
        c = a + b;
        a = b;
        b = c;
    }
    printf("\n");
}

void factorial() {
    int n = 5, f = 1;
    for (int i = 1; i <= n; i++)
        f *= i;
    printf("Child 2 (Factorial of 5): %d\n", f);
}

void print_even() {
    printf("Child 3 (Even numbers): ");
    for (int i = 0; i <= 10; i += 2)
        printf("%d ", i);
```

```c
    printf("\n");
}

void print_odd() {
    printf("Child 4 (Odd numbers): ");
    for (int i = 1; i <= 10; i += 2)
        printf("%d ", i);
    printf("\n");
}

void show_pid() {
    printf("Child 5 (PID): %d\n", getpid());
}

int main() {
    for (int i = 1; i <= 5; i++) {
        pid_t pid = fork();

        if (pid < 0) {
            perror("Fork failed");
            exit(1);
        }

        else if (pid == 0) {
            // CHILD PROCESS
            switch (i) {
                case 1: fibonacci(); break;
                case 2: factorial(); break;
                case 3: print_even(); break;
                case 4: print_odd(); break;
                case 5: show_pid(); break;
            }
            exit(0);
        }
    }

    // PARENT waits for all children
    for (int i = 0; i < 5; i++)
        wait(NULL);

    printf("Parent: All child processes finished.\n");

    return 0;
}
```

29. Implement a program using vfork() where the child reads a login name and the parent reads the password.

```
BEGIN VFORK_LOGIN_PASSWORD

   DEFINE global/shared buffer: char login[50], password[50]

   CALL pid = vfork()

   IF pid < 0:
      PRINT error and terminate

   ELSE IF pid == 0:      // CHILD PROCESS
      vfork RULES:
         - parent is suspended
         - child and parent share same address space
         - child MUST NOT return from main or call exit()
         - must call _exit()

      PROMPT "Enter login name: "
      READ input into login[]
      CALL _exit(0) to resume parent

   ELSE                  // PARENT EXECUTES AFTER CHILD EXITS
      PROMPT "Enter password: "
      READ input into password[]

      PRINT login and password OR process authentication

   END IF

END VFORK_LOGIN_PASSWORD
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
   char login[50];
   char password[50];

   pid_t pid = vfork();

   if (pid < 0) {
      perror("vfork failed");
```

```
        exit(1);
    }

    else if (pid == 0) {   // child
        printf("Enter login name: ");
        scanf("%s", login);

        // MUST use _exit() with vfork
        _exit(0);
    }

    else {   // parent resumes only after child _exit()
        printf("Enter password: ");
        scanf("%s", password);

        printf("\n--- Credentials Entered ---\n");
        printf("Login Name : %s\n", login);
        printf("Password   : %s\n", password);
    }

    return 0;
}
```

30. Write a program that launches an application using the fork() system call.

```
BEGIN LAUNCH_APPLICATION_USING_FORK

    1. CALL fork() → returns pid

    2. IF pid < 0:
        PRINT "fork failed"
        TERMINATE program

    3. ELSE IF pid == 0:    // CHILD PROCESS
        PREPARE arguments for the application (argv[] array)
        CALL execvp(program_name, argv)
        IF execvp fails:
            PRINT error message
            CALL exit(1)

    4. ELSE (pid > 0):      // PARENT PROCESS
        PRINT "Parent: Child process created"
        OPTIONALLY wait() for child to finish
        CONTINUE execution OR exit
```

END LAUNCH_APPLICATION_USING_FORK

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

int main() {
    pid_t pid = fork();

    if (pid < 0) {
        perror("fork failed");
        exit(1);
    }

    else if (pid == 0) {
        // CHILD: launch application
        char *app = "gedit";   // change this to "firefox", "vlc", etc.
        char *args[] = {app, NULL};

        printf("Launching %s...\n", app);

        execvp(app, args);  // execute the application

        perror("exec failed"); // executes only if exec fails
        exit(1);
    }

    else {
        // PARENT
        printf("Parent: waiting for child to exit...\n");
        wait(NULL);
        printf("Parent: child process finished.\n");
    }

    return 0;
}
```