A1

1)
```
#include <iostream>
#include <unistd.h>     // for pipe, fork, read, write
#include <sys/types.h>
#include <sys/wait.h>   // for wait

int main() {
    int pipe1[2], pipe2[2];
    // Create two pipes: pipe1 for parent->child, pipe2 for child->parent
    if (pipe(pipe1) == -1 || pipe(pipe2) == -1) {
        std::cerr << "Pipe creation failed\n";
        return 1;
    }

    pid_t pid = fork();
    if (pid < 0) {
        std::cerr << "Fork failed\n";
        return 1;
    } else if (pid > 0) {
        // Parent process
        close(pipe1[0]); // Close unused read end
        close(pipe2[1]); // Close unused write end

        char input[100];
        std::cout << "Enter a string: ";
        std::cin.getline(input, 100);

        // Compute length manually (no string functions)
        int len = 0;
        while (input[len] != '\0') {
            len++;
        }

        // Send string (including null terminator) to child
        write(pipe1[1], input, len + 1);

        // Read concatenated result from child
        char result[200];
        read(pipe2[0], result, sizeof(result));
        std::cout << "Concatenated string: " << result << std::endl;

        close(pipe1[1]);
        close(pipe2[0]);
        wait(NULL); // Wait for child to finish
    } else {
        // Child process
        close(pipe1[1]); // Close unused write end
        close(pipe2[0]); // Close unused read end

        // Read input from parent
        char received[100];
        read(pipe1[0], received, sizeof(received));

        // String to concatenate
        char extra[] = " - concatenated by child";

        // Manual concatenation
        char result[200];
```

```
        int i = 0;
        while (received[i] != '\0') {
            result[i] = received[i];
            i++;
        }
        int j = 0;
        while (extra[j] != '\0') {
            result[i + j] = extra[j];
            j++;
        }
        result[i + j] = '\0';

        // Send back to parent
        write(pipe2[1], result, i + j + 1);

        close(pipe1[0]);
        close(pipe2[1]);
    }

    return 0;
}
```

2)

```
#include <iostream>
#include <unistd.h>      // for pipe, fork, read, write
#include <sys/types.h>
#include <sys/wait.h>    // for wait
#include <cstdlib>       // for exit

int main() {
    int pipe1[2], pipe2[2];
    // pipe1: parent -> child, pipe2: child -> parent
    if (pipe(pipe1) == -1 || pipe(pipe2) == -1) {
        std::cerr << "Failed to create pipes\n";
        return 1;
    }

    pid_t pid = fork();
    if (pid < 0) {
        std::cerr << "Fork failed\n";
        return 1;
    }

    if (pid > 0) {
        // Parent process
        close(pipe1[0]); // close read end of pipe1
        close(pipe2[1]); // close write end of pipe2

        int rows, cols;
        std::cout << "Enter number of rows: ";
        std::cin >> rows;
        std::cout << "Enter number of columns: ";
        std::cin >> cols;

        int size = rows * cols;
        int* A = new int[size];
        int* B = new int[size];

        std::cout << "Enter elements of first matrix:\n";
        for (int i = 0; i < size; ++i) {
```

```
        std::cin >> A[i];
    }
    std::cout << "Enter elements of second matrix:\n";
    for (int i = 0; i < size; ++i) {
        std::cin >> B[i];
    }

    // Send rows and cols
    write(pipe1[1], &rows, sizeof(rows));
    write(pipe1[1], &cols, sizeof(cols));
    // Send matrix data
    write(pipe1[1], A, size * sizeof(int));
    write(pipe1[1], B, size * sizeof(int));

    delete[] A;
    delete[] B;
    close(pipe1[1]);

    // Read result dims (optional)
    int r2, c2;
    read(pipe2[0], &r2, sizeof(r2));
    read(pipe2[0], &c2, sizeof(c2));
    int size2 = r2 * c2;
    int* C = new int[size2];
    read(pipe2[0], C, size2 * sizeof(int));

    std::cout << "Sum matrix is:\n";
    for (int i = 0; i < r2; ++i) {
        for (int j = 0; j < c2; ++j) {
            std::cout << C[i * c2 + j] << " ";
        }
        std::cout << "\n";
    }

    delete[] C;
    close(pipe2[0]);
    wait(NULL);
} else {
    // Child process
    close(pipe1[1]); // close write end of pipe1
    close(pipe2[0]); // close read end of pipe2

    int rows, cols;
    // Read dims
    read(pipe1[0], &rows, sizeof(rows));
    read(pipe1[0], &cols, sizeof(cols));
    int size = rows * cols;
    int* A = new int[size];
    int* B = new int[size];
    // Read matrices
    read(pipe1[0], A, size * sizeof(int));
    read(pipe1[0], B, size * sizeof(int));
    close(pipe1[0]);

    int* C = new int[size];
    for (int i = 0; i < size; ++i) {
        C[i] = A[i] + B[i];
    }

    // Send back dims and summed matrix
```

```
        write(pipe2[1], &rows, sizeof(rows));
        write(pipe2[1], &cols, sizeof(cols));
        write(pipe2[1], C, size * sizeof(int));

        delete[] A;
        delete[] B;
        delete[] C;
        close(pipe2[1]);
    }

    return 0;
}
```

A2

2) server

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

// Shared data layout
typedef struct {
    sem_t sem;    // POSIX unnamed semaphore
    int counter;  // shared counter
} shared_data_t;

int main() {
    const char *path = "/tmp/my_shm";
    int fd = open(path, O_RDWR | O_CREAT, 0666);
    if (fd == -1) { perror("open"); exit(EXIT_FAILURE); }
    if (ftruncate(fd, sizeof(shared_data_t)) == -1) { perror("ftruncate"); exit(EXIT_FAILURE); }

    shared_data_t *shm = mmap(
        NULL,
        sizeof(shared_data_t),
        PROT_READ | PROT_WRITE,
        MAP_SHARED,
        fd,
        0
    );
    if (shm == MAP_FAILED) { perror("mmap"); exit(EXIT_FAILURE); }

    // Initialize semaphore for inter-process use, and counter
    sem_init(&shm->sem, 1, 1);
    shm->counter = 0;

    printf("[Server] Ready. Press Enter to increment counter, 'q'+Enter to quit.\n");
    char buf[8];
    while (fgets(buf, sizeof(buf), stdin)) {
        if (buf[0] == 'q') break;

        // Critical section
        sem_wait(&shm->sem);
        shm->counter++;
        printf("[Server] Counter = %d\n", shm->counter);
```

```
        sem_post(&shm->sem);
    }

    // Cleanup
    sem_destroy(&shm->sem);
    munmap(shm, sizeof(shared_data_t));
    close(fd);
    printf("[Server] Exiting.\n");
    return 0;
}
```

3) client

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

typedef struct {
    sem_t sem;
    int counter;
} shared_data_t;

int main() {
    const char *path = "/tmp/my_shm";
    int fd = open(path, O_RDWR);
    if (fd == -1) { perror("open"); exit(EXIT_FAILURE); }

    shared_data_t *shm = mmap(
        NULL,
        sizeof(shared_data_t),
        PROT_READ | PROT_WRITE,
        MAP_SHARED,
        fd,
        0
    );
    if (shm == MAP_FAILED) { perror("mmap"); exit(EXIT_FAILURE); }

    printf("[Client] Ready. Press Enter to read counter, 'q'+Enter to quit.\n");
    char buf[8];
    while (fgets(buf, sizeof(buf), stdin)) {
        if (buf[0] == 'q') break;

        sem_wait(&shm->sem);
        printf("[Client] Counter = %d\n", shm->counter);
        sem_post(&shm->sem);
    }

    munmap(shm, sizeof(shared_data_t));
    close(fd);
    printf("[Client] Exiting.\n");
    return 0;
}
```

1)doubt

A3

1)server

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

#define BUF_SIZE 1024

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <port>\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    int port = atoi(argv[1]);

    // 1) Create socket
    int sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    // 2) Bind to all interfaces on given port
    struct sockaddr_in srv_addr = {0};
    srv_addr.sin_family = AF_INET;
    srv_addr.sin_addr.s_addr = INADDR_ANY;
    srv_addr.sin_port = htons(port);

    if (bind(sock, (struct sockaddr*)&srv_addr, sizeof(srv_addr)) < 0) {
        perror("bind");
        close(sock);
        exit(EXIT_FAILURE);
    }

    printf("UDP server listening on port %d\n", port);

    // 3) Receive loop
    char buf[BUF_SIZE];
    struct sockaddr_in cli_addr;
    socklen_t cli_len = sizeof(cli_addr);

    while (1) {
        ssize_t n = recvfrom(sock, buf, BUF_SIZE-1, 0,
                             (struct sockaddr*)&cli_addr, &cli_len);
        if (n < 0) {
            perror("recvfrom");
            break;
        }
        buf[n] = '\0';
        printf("Received from %s:%d: \"%s\"\n",
               inet_ntoa(cli_addr.sin_addr),
               ntohs(cli_addr.sin_port),
               buf);
        // (Optionally) send a reply:
        // sendto(sock, buf, n, 0, (struct sockaddr*)&cli_addr, cli_len);
    }

    close(sock);
```

```
    return 0;
}
```

client

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

#define BUF_SIZE 1024

int main(int argc, char *argv[]) {
    if (argc != 4) {
        fprintf(stderr, "Usage: %s <server-ip> <server-port> <message>\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    const char *srv_ip   = argv[1];
    int         srv_port = atoi(argv[2]);
    const char *msg      = argv[3];

    // 1) Create socket
    int sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    // 2) Fill in server address
    struct sockaddr_in srv_addr = {0};
    srv_addr.sin_family = AF_INET;
    srv_addr.sin_port   = htons(srv_port);
    if (inet_aton(srv_ip, &srv_addr.sin_addr) == 0) {
        fprintf(stderr, "Invalid IP address: %s\n", srv_ip);
        close(sock);
        exit(EXIT_FAILURE);
    }

    // 3) Send the message
    ssize_t sent = sendto(sock, msg, strlen(msg), 0,
                          (struct sockaddr*)&srv_addr, sizeof(srv_addr));
    if (sent < 0) {
        perror("sendto");
        close(sock);
        exit(EXIT_FAILURE);
    }

    printf("Sent \"%s\" to %s:%d\n", msg, srv_ip, srv_port);

    // (Optionally) receive a reply:
    // char buf[BUF_SIZE];
    // struct sockaddr_in from;
    // socklen_t fromlen = sizeof(from);
    // ssize_t n = recvfrom(sock, buf, BUF_SIZE-1, 0, (struct sockaddr*)&from, &fromlen);
    // if (n>0) { buf[n]='\0'; printf("Reply: %s\n", buf); }

    close(sock);
    return 0;
```

```
}
```

2)

```c
// file: mpi_master_worker.c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

#define TAG_STOP   0
#define TAG_TASK   1
#define TAG_RESULT 2

// Define your task and result payloads:
typedef struct {
    long start;     // inclusive
    long end;       // exclusive
} task_t;

typedef struct {
    double partial_sum;
    long  start;
    long  end;
} result_t;

// Example heavy-compute function
double f(long i) {
    // Just a dummy; replace with your real work.
    return 1.0 / (1.0 + i*i);
}

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, P;
    MPI_Comm_size(MPI_COMM_WORLD, &P);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    const long N = 100000000L;       // total elements
    const long CHUNK_SIZE = 100000;  // elements per task

    if (rank == 0) {
        // ---------------- Master ----------------
        long num_tasks = (N + CHUNK_SIZE - 1) / CHUNK_SIZE;
        long next_task = 0;
        double total = 0.0;

        // 1) Prime the workers with one task each:
        for (int worker = 1; worker < P && next_task < num_tasks; worker++) {
            task_t t;
            t.start = next_task * CHUNK_SIZE;
            t.end   = ((next_task+1)*CHUNK_SIZE < N ? (next_task+1)*CHUNK_SIZE : N);
            MPI_Send(&t, sizeof(t), MPI_BYTE,
                     worker, TAG_TASK, MPI_COMM_WORLD);
            next_task++;
        }

        // 2) Loop: receive results, send new tasks
        for (long completed = 0; completed < num_tasks; completed++) {
            result_t r;
            MPI_Status st;
```

```
            MPI_Recv(&r, sizeof(r), MPI_BYTE,
                     MPI_ANY_SOURCE, TAG_RESULT,
                     MPI_COMM_WORLD, &st);
            total += r.partial_sum;

            int worker = st.MPI_SOURCE;
            // If there’s more work, send the next task:
            if (next_task < num_tasks) {
                task_t t;
                t.start = next_task * CHUNK_SIZE;
                t.end   = ((next_task+1)*CHUNK_SIZE < N ? (next_task+1)*CHUNK_SIZE : N);
                MPI_Send(&t, sizeof(t), MPI_BYTE,
                         worker, TAG_TASK, MPI_COMM_WORLD);
                next_task++;
            }
            else {
                // Tell that worker to stop
                MPI_Send(NULL, 0, MPI_BYTE,
                         worker, TAG_STOP, MPI_COMM_WORLD);
            }
        }

        printf("Final sum = %.6f\n", total);
    }
    else {
        // ---------------- Worker ----------------
        while (1) {
            MPI_Status st;
            // Probe for an incoming task or stop tag
            MPI_Probe(0, MPI_ANY_TAG, MPI_COMM_WORLD, &st);
            if (st.MPI_TAG == TAG_STOP) {
                // Clean up and exit
                MPI_Recv(NULL, 0, MPI_BYTE, 0, TAG_STOP, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
                break;
            }
            // Otherwise it’s a real task
            task_t t;
            MPI_Recv(&t, sizeof(t), MPI_BYTE, 0, TAG_TASK, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

            // Do the work
            result_t r;
            r.partial_sum = 0.0;
            r.start = t.start;
            r.end   = t.end;
            for (long i = t.start; i < t.end; i++) {
                r.partial_sum += f(i);
            }

            // Send result back
            MPI_Send(&r, sizeof(r), MPI_BYTE,
                     0, TAG_RESULT, MPI_COMM_WORLD);
        }
    }

    MPI_Finalize();
    return 0;
}
```

```
mpicc -O2 -o mpi_master_worker mpi_master_worker.c
mpirun -np 8 ./mpi_master_worker
```

A4

1)
Server

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

#define PORT     12345
#define BACKLOG    5
#define BUF_SIZE  64

/* Compute n! (note: long long overflows above ~20) */
static long long factorial(int n) {
    long long f = 1;
    int i;
    for (i = 2; i <= n; i++) {
        f *= i;
    }
    return f;
}

int main(void) {
    int listen_fd, conn_fd, opt, n;
    struct sockaddr_in srv_addr, cli_addr;
    socklen_t cli_len;
    uint32_t net_n;
    ssize_t r;
    char buf[BUF_SIZE];
    int blen;
    long long result;

    /* 1) create listening socket */
    listen_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (listen_fd < 0) {
        perror("socket");
        exit(1);
    }

    /* allow address reuse */
    opt = 1;
    setsockopt(listen_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

    /* bind to all interfaces */
    memset(&srv_addr, 0, sizeof(srv_addr));
    srv_addr.sin_family = AF_INET;
    srv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    srv_addr.sin_port = htons(PORT);

    if (bind(listen_fd, (struct sockaddr*)&srv_addr, sizeof(srv_addr)) < 0) {
        perror("bind");
```

```
        close(listen_fd);
        exit(1);
    }

    if (listen(listen_fd, BACKLOG) < 0) {
        perror("listen");
        close(listen_fd);
        exit(1);
    }

    printf("Server listening on port %d\n", PORT);

    for (;;) {
        cli_len = sizeof(cli_addr);
        conn_fd = accept(listen_fd, (struct sockaddr*)&cli_addr, &cli_len);
        if (conn_fd < 0) {
            perror("accept");
            continue;
        }

        /* 2) receive 32-bit integer n */
        r = recv(conn_fd, &net_n, sizeof(net_n), MSG_WAITALL);
        if (r != sizeof(net_n)) {
            fprintf(stderr, "recv failed\n");
            close(conn_fd);
            continue;
        }
        n = (int)ntohl(net_n);
        printf("Received n=%d from %s:%d\n",
               n,
               inet_ntoa(cli_addr.sin_addr),
               ntohs(cli_addr.sin_port));

        /* 3) compute factorial */
        result = factorial(n);

        /* 4) format result as ASCII string */
        blen = snprintf(buf, BUF_SIZE, "%lld", result);
        if (blen < 0 || blen >= BUF_SIZE) {
            fprintf(stderr, "snprintf error\n");
            close(conn_fd);
            continue;
        }

        /* 5) send NUL-terminated string */
        send(conn_fd, buf, blen+1, 0);

        close(conn_fd);
    }

    /* unreachable */
    close(listen_fd);
    return 0;
}
```

client

```
/* client.c */
#include <stdio.h>
#include <stdlib.h>
```

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

#define PORT      12345
#define BUF_SIZE   64

int main(int argc, char *argv[]) {
    int sock, n;
    struct sockaddr_in srv_addr;
    char buf[BUF_SIZE];
    ssize_t r;
    uint32_t net_n;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s <server-ip> <n>\n", argv[0]);
        return 1;
    }
    n = atoi(argv[2]);

    /* 1) create socket */
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        perror("socket");
        return 1;
    }

    /* 2) configure server address */
    memset(&srv_addr, 0, sizeof(srv_addr));
    srv_addr.sin_family = AF_INET;
    srv_addr.sin_port = htons(PORT);
    if (inet_aton(argv[1], &srv_addr.sin_addr) == 0) {
        fprintf(stderr, "Invalid IP: %s\n", argv[1]);
        close(sock);
        return 1;
    }

    /* 3) connect */
    if (connect(sock, (struct sockaddr*)&srv_addr, sizeof(srv_addr)) < 0) {
        perror("connect");
        close(sock);
        return 1;
    }

    /* 4) send n in network byte order */
    net_n = htonl((uint32_t)n);
    if (send(sock, &net_n, sizeof(net_n), 0) != sizeof(net_n)) {
        perror("send");
        close(sock);
        return 1;
    }

    /* 5) receive NUL-terminated ASCII factorial */
    r = recv(sock, buf, BUF_SIZE, MSG_WAITALL);
    if (r <= 0) {
        fprintf(stderr, "recv failed\n");
        close(sock);
        return 1;
    }
```

```
    buf[BUF_SIZE-1] = '\0';  /* ensure NUL termination */

    printf("Factorial of %d is %s\n", n, buf);
    close(sock);
    return 0;
}
```

2)
mapper.c

```
/* mapper.c */
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <sys/stat.h>
#include <pwd.h>
#include <string.h>
#include <unistd.h>

int main(void) {
    DIR *d = opendir(".");
    if (!d) {
        perror("opendir");
        return 1;
    }

    struct dirent *ent;
    struct stat st;
    while ((ent = readdir(d)) != NULL) {
        /* skip "." and ".." */
        if (strcmp(ent->d_name, ".") == 0 || strcmp(ent->d_name, "..") == 0)
            continue;

        if (stat(ent->d_name, &st) < 0) {
            /* could not stat—skip */
            continue;
        }
        /* only regular files */
        if (!S_ISREG(st.st_mode))
            continue;

        /* lookup owner name */
        struct passwd *pw = getpwuid(st.st_uid);
        const char *user = pw ? pw->pw_name : "UNKNOWN";

        /* emit: size \t user */
        printf("%lld\t%s\n", (long long)st.st_size, user);
    }

    closedir(d);
    return 0;
}
gcc -O2 -o mapper mapper.c
```

reducer.c

```
/* reducer.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#define MAX_USERS 1024
#define MAX_USER_LEN 256

int main(void) {
    char line[1024];
    long long max_size = -1;
    char users[MAX_USERS][MAX_USER_LEN];
    int user_count = 0;

    while (fgets(line, sizeof(line), stdin)) {
        long long sz;
        char user[MAX_USER_LEN];
        if (sscanf(line, "%lld\t%255s", &sz, user) != 2)
            continue;

        if (sz > max_size) {
            /* new maximum: reset list */
            max_size = sz;
            user_count = 0;
            strncpy(users[user_count++], user, MAX_USER_LEN-1);
            users[user_count-1][MAX_USER_LEN-1] = '\0';
        }
        else if (sz == max_size) {
            /* add if not already in list */
            int found = 0;
            for (int i = 0; i < user_count; i++) {
                if (strcmp(users[i], user) == 0) {
                    found = 1;
                    break;
                }
            }
            if (!found && user_count < MAX_USERS) {
                strncpy(users[user_count++], user, MAX_USER_LEN-1);
                users[user_count-1][MAX_USER_LEN-1] = '\0';
            }
        }
    }

    /* output the owners of largest files */
    for (int i = 0; i < user_count; i++) {
        printf("%s\n", users[i]);
    }
    return 0;
}
gcc -O2 -o reducer reducer.c
```

A5

```
fle_transfer.x
program FILE_TRANSFER_PROG {
    version FILE_TRANSFER_VERS {
        /* get_file: client->server filename (string), server->client file data (opaque) */
        getfile STRING -> FILEDATA  = 1;
        /* put_file: client->server filename + data, server->client status (int) */
        putfile PUTARGS -> int      = 2;
    } = 1;
} = 0x20000001;

/* Data types */
typedef string STRING<255>;
```

```
struct FILEDATA {
    unsigned long size;
    opaque data<>;
};

struct PUTARGS {
    STRING filename;
    unsigned long size;
    opaque data<>;
};
rpcgen -a file_transfer.x
```

file_transfer_svc.c

```
#include "file_transfer.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <rpc/rpc.h>

/* Handler for getfile_1 */
FILEDATA *
getfile_1_svc(STRING *filename, struct svc_req *rqstp)
{
    static FILEDATA result;
    FILE *f;
    unsigned long n;

    f = fopen(filename->str, "rb");
    if (!f) {
        result.size = 0;
        result.data.data_len = 0;
        return &result;
    }
    fseek(f, 0, SEEK_END);
    n = ftell(f);
    rewind(f);

    result.size = n;
    result.data.data_len = n;
    result.data.data_val = malloc(n);
    fread(result.data.data_val, 1, n, f);
    fclose(f);
    return &result;
}

/* Handler for putfile_1 */
int *
putfile_1_svc(PUTARGS *args, struct svc_req *rqstp)
{
    static int status;
    FILE *f;

    f = fopen(args->filename.filename, "wb");
```

```
    if (!f) {
        status = -1;
        return &status;
    }
    fwrite(args->data.data_val, 1, args->size, f);
    fclose(f);
    status = 0;
    return &status;
}
```

file_transfer_clnt.c

```
#include "file_transfer.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void usage(const char *prog) {
    fprintf(stderr,
        "Usage: %s <server> get <filename> | put <localfile> <remotefile>\n",
        prog);
    exit(1);
}

int main(int argc, char **argv) {
    CLIENT *cl;
    char *host;

    if (argc < 4) usage(argv[0]);
    host = argv[1];
    cl = clnt_create(host, FILE_TRANSFER_PROG, FILE_TRANSFER_VERS, "tcp");
    if (!cl) {
        clnt_pcreateerror(host);
        exit(1);
    }

    if (strcmp(argv[2], "get") == 0) {
        STRING filename;
        FILEDATA *resp;
        filename.str = argv[3];
        filename.str_len = strlen(argv[3]);
        resp = getfile_1(&filename, cl);
        if (resp && resp->size > 0) {
            fwrite(resp->data.data_val, 1, resp->size, stdout);
        } else {
            fprintf(stderr, "Error: could not fetch '%s'\n", argv[3]);
        }
    }
    else if (strcmp(argv[2], "put") == 0 && argc==5) {
        PUTARGS args;
        int *rstatus;
        FILE *f = fopen(argv[3], "rb");
        if (!f) { perror("fopen"); exit(1);}
```

```
        fseek(f,0,SEEK_END); args.size = ftell(f); rewind(f);
        args.data.data_len = args.size;
        args.data.data_val = malloc(args.size);
        fread(args.data.data_val,1,args.size,f);
        fclose(f);
        args.filename.filename = argv[4];
        args.filename.filename_len = strlen(argv[4]);

        rstatus = putfile_1(&args, cl);
        if (!rstatus || *rstatus!=0)
            fprintf(stderr, "Error: putfile failed\n");
    }
    else {
        usage(argv[0]);
    }

    clnt_destroy(cl);
    return 0;
}
```

A7

1)
a)

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

#define TAG_REQ   1
#define TAG_ACK   2
#define TAG_REL   3

int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);
    int P, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &P);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // Lamport clock
    int clock = 0;
    // Highest timestamp of our own request (if pending), else -1
    int req_ts = -1;
    // Count of outstanding ACKs
    int pending_acks = 0;
    // Deferred requests queue: we only need to know which ranks to ACK later
    int deferred[128], defcnt = 0;

    // For demonstration, each process enters CS exactly once, after a barrier
    MPI_Barrier(MPI_COMM_WORLD);
    // 1) Request CS
    clock++;
```

```
    req_ts = clock;
    pending_acks = P-1;
    // broadcast request(ts, rank)
    for (int dst = 0; dst < P; dst++) if (dst != rank) {
        MPI_Send(&req_ts, 1, MPI_INT, dst, TAG_REQ, MPI_COMM_WORLD);
    }
    printf("[R%d] Requested CS at ts=%d, waiting %d acks\n", rank, req_ts, pending_acks);

    // 2) Process incoming messages until we collect all ACKs
    while (pending_acks > 0) {
        MPI_Status st;
        int in_ts;
        MPI_Recv(&in_ts, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD, &st);
        clock = clock > in_ts ? clock+1 : in_ts+1;

        if (st.MPI_TAG == TAG_REQ) {
            int src = st.MPI_SOURCE;
            // if our own request is earlier, defer; else ack immediately
            if (req_ts >= 0 && (req_ts < in_ts || (req_ts == in_ts && rank < src))) {
                deferred[defcnt++] = src;
            } else {
                MPI_Send(&clock, 1, MPI_INT, src, TAG_ACK, MPI_COMM_WORLD);
            }
        }
        else if (st.MPI_TAG == TAG_ACK) {
            pending_acks--;
        }
    }

    // 3) ENTER Critical Section
    printf("[R%d] ENTER CS at ts=%d\n", rank, clock);
    sleep(1);                  // simulate work
    printf("[R%d] LEAVE CS at ts=%d\n", rank, clock);

    // 4) RELEASE: send release to deferred
    for (int i = 0; i < defcnt; i++) {
        int dst = deferred[i];
        MPI_Send(&clock, 1, MPI_INT, dst, TAG_ACK, MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}
mpicc -O2 -o lamport lamport.c
mpirun -np 5 ./lamport
```

b)

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

```
#define TAG_REQ   1
#define TAG_REPLY 2

int main(int argc, char **argv) {
   MPI_Init(&argc, &argv);
   int P, rank;
   MPI_Comm_size(MPI_COMM_WORLD, &P);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   int clock = 0;
   int req_ts = -1, replies_needed = 0;
   int deferred[128], defcnt = 0;

   MPI_Barrier(MPI_COMM_WORLD);

   // 1) send REQUEST to all
   clock++;
   req_ts = clock;
   replies_needed = P-1;
   for (int dst = 0; dst < P; dst++) if (dst != rank) {
      MPI_Send(&req_ts, 1, MPI_INT, dst, TAG_REQ, MPI_COMM_WORLD);
   }
   printf("[R%d] Req CS ts=%d\n", rank, req_ts);

   // 2) wait for replies
   while (replies_needed > 0) {
      MPI_Status st;
      int in_ts;
      MPI_Recv(&in_ts, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD, &st);
      clock = clock > in_ts ? clock+1 : in_ts+1;

      if (st.MPI_TAG == TAG_REQ) {
         int src = st.MPI_SOURCE;
         // defer if our request is earlier, else reply immediately
         if (req_ts >= 0 && (req_ts < in_ts || (req_ts == in_ts && rank < src))) {
            deferred[defcnt++] = src;
         } else {
            MPI_Send(&clock, 1, MPI_INT, src, TAG_REPLY, MPI_COMM_WORLD);
         }
      }
      else if (st.MPI_TAG == TAG_REPLY) {
         replies_needed--;
      }
   }

   // 3) Critical Section
   printf("[R%d] ENTER CS ts=%d\n", rank, clock);
   sleep(1);
   printf("[R%d] LEAVE CS ts=%d\n", rank, clock);
```

```
    // 4) reply to deferred
    for (int i = 0; i < defcnt; i++) {
        MPI_Send(&clock, 1, MPI_INT, deferred[i], TAG_REPLY, MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}
mpicc -O2 -o ricart ricart.c
mpirun -np 5 ./ricart
```

2)

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define TAG_REPORT  1
#define TAG_DETECT  2

// Max processes and edges
#define MAXP 64
#define MAXE 128

// A single edge: process 'u' is waiting for process 'v'
typedef struct {
    int u, v;
} edge_t;

// Each worker reports its edges in a variable-length array
typedef struct {
    int count;
    edge_t edges[MAXE];
} wfg_t;

// coordinator: detect cycle in directed graph
int has_cycle(int P, int adj[][MAXP]) {
    int visited[MAXP] = {0}, stack[MAXP] = {0};
    // DFS from each node
    for (int i = 1; i < P; i++) {
        memset(visited, 0, sizeof(visited));
        memset(stack, 0, sizeof(stack));
        // inner recursive lambda simulated by explicit stack:
        int st_sz = 0;
        // push (i, next_child=1)
        int node = i, next = 1;
        int call_stack[MAXP][2];
        call_stack[st_sz][0] = node; call_stack[st_sz][1] = next;
        st_sz++; visited[node] = 1; stack[node] = 1;
        while (st_sz) {
            node = call_stack[st_sz-1][0];
```

```
            next = call_stack[st_sz-1][1];
            if (next >= P) {
                // done with node
                stack[node] = 0;
                st_sz--;
                continue;
            }
            call_stack[st_sz-1][1]++;  // next time, try child+1
            if (adj[node][next]) {
                if (stack[next]) return 1;
                if (!visited[next]) {
                    // push child
                    visited[next] = 1; stack[next] = 1;
                    call_stack[st_sz][0] = next;
                    call_stack[st_sz][1] = 1;
                    st_sz++;
                }
            }
        }
    }
    return 0;
}

int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);
    int P, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &P);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        // coordinator
        int adj[MAXP][MAXP];
        memset(adj, 0, sizeof(adj));
        wfg_t report;
        MPI_Status st;

        while (1) {
            // 1) ask all workers to report
            for (int src = 1; src < P; src++) {
                MPI_Send(NULL, 0, MPI_BYTE, src, TAG_DETECT, MPI_COMM_WORLD);
            }
            // 2) gather
            memset(adj, 0, sizeof(adj));
            for (int src = 1; src < P; src++) {
                MPI_Recv(&report, sizeof(report), MPI_BYTE, src,
                         TAG_REPORT, MPI_COMM_WORLD, &st);
                for (int e = 0; e < report.count; e++) {
                    adj[ report.edges[e].u ][ report.edges[e].v ] = 1;
                }
            }
            // 3) detect
            if (has_cycle(P, adj)) {
```

```c
            printf("[COORD] Deadlock detected!\n");
        } else {
            printf("[COORD] No deadlock.\n");
        }
        fflush(stdout);
        sleep(5);  // periodic
    }
  }
  else {
    // worker: simulate holding/waiting
    wfg_t local;
    MPI_Status st;

    // for demo, each worker cycles through some fixed edges
    while (1) {
        // wait for detect request
        MPI_Recv(NULL, 0, MPI_BYTE, 0, TAG_DETECT, MPI_COMM_WORLD, &st);

        // build local wait-for graph edges
        // e.g. rank i waits for (i%P)+1
        local.count = 1;
        local.edges[0].u = rank;
        local.edges[0].v = (rank % (P-1)) + 1;

        // send report
        MPI_Send(&local, sizeof(local), MPI_BYTE, 0,
                 TAG_REPORT, MPI_COMM_WORLD);
    }
  }

  MPI_Finalize();
  return 0;
}
mpicc -O2 -o deadlock deadlock.c
# Centralized deadlock detection with 5 ranks (1 coordinator + 4 workers):
mpirun -np 5 ./deadlock
```

A