# **Systems Programming (6CC514), 2024/25**

Blue text on a black background

Description automatically generated A logo on a black background

Description automatically generated

**100667771/2024-25**

## Assessment 2 (A2). Systems programming in C using Linux system calls. Inter process communication (IPC).

3. Utility program with command line interface (W7)

Exercise: Compose a utility program with command line interface to provide the required service, implement at list 3 options (keys). Organize processing of information required by the task variant as a C language function. Work with arrays of characters and pointers directly. Use neither standard libc functions for string processing nor other libraries.

nano replace\_substring.c

gcc replace\_substring.c -o replace\_substring

./replace\_substring

**replace\_substring.c**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

void replace\_substring(const char\* input, const char\* find, const char\* replace, char\* output, size\_t max\_len) {

const char\* pos = input;

size\_t find\_len = strlen(find);

size\_t replace\_len = strlen(replace);

size\_t out\_index = 0;

while (\*pos && out\_index < max\_len - 1) {

const char\* match = strstr(pos, find);

if (match) {

size\_t chunk\_size = match - pos;

if (out\_index + chunk\_size >= max\_len - 1) break;

strncpy(output + out\_index, pos, chunk\_size);

output[out\_index + chunk\_size] = '\0';

out\_index += chunk\_size;

if (out\_index + replace\_len >= max\_len - 1) break;

strncpy(output + out\_index, replace, replace\_len);

output[out\_index + replace\_len] = '\0';

out\_index += replace\_len;

pos = match + find\_len;

} else {

size\_t remaining\_len = strlen(pos);

if (out\_index + remaining\_len >= max\_len - 1) break;

strncpy(output + out\_index, pos, remaining\_len);

out\_index += remaining\_len;

break;

}

}

output[out\_index] = '\0';

}

int main() {

char input[1024];

char find[1024];

char replace[1024];

char output[1024];

size\_t max\_len = sizeof(output);

printf("Enter the input string: ");

fgets(input, sizeof(input), stdin);

input[strcspn(input, "\n")] = '\0';

printf("Enter the substring to find: ");

fgets(find, sizeof(find), stdin);

find[strcspn(find, "\n")] = '\0';

printf("Enter the replacement substring: ");

fgets(replace, sizeof(replace), stdin);

replace[strcspn(replace, "\n")] = '\0';

replace\_substring(input, find, replace, output, max\_len);

printf("Original string: %s\n", input);

printf("Modified string: %s\n", output);

return 0;

}

4. IPC via pipes (W8)

Exercise: Please implement the required information processing using a pair of specialized processes – client and server; provide the information communication between the processes via Linux pipes.

* Use the data processing task of W1.

**Filename:** server.c

void process\_data(const char\* input, char\* output) {

snprintf(output, 1024, "Processed: %s", input);

}

**Explanation:**

This function simulates the processing task by appending "Processed:" to the input string.It is called within the server to handle data sent from the client.

* Compose a client that periodically inputs data and sends it to the server

**Filename:** client.c

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

int main() {

char input[1024];

char response[1024];

int pipe1[2], pipe2[2];

printf("Client started\n");

while (1) {

printf("Enter data to send to the server (or type 'exit' to quit): ");

fgets(input, sizeof(input), stdin);

input[strcspn(input, "\n")] = '\0';

write(pipe1[1], input, strlen(input) + 1);

if (strcmp(input, "exit") == 0) break;

read(pipe2[0], response, sizeof(response));

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

}

close(pipe1[1]);

close(pipe2[0]);

return 0;

}

**Explanation:**

The client collects input from the user and sends it to the server through pipe1.It waits for a response from the server through pipe2 and prints the result.The loop exits when the user inputs "exit."

* Compose a server that receives data from the client, processes it, and returns the result back to the client.

**Filename:** server.c

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

void process\_data(const char\* input, char\* output);

int main() {

char buffer[1024];

char response[1024];

int pipe1[2], pipe2[2];

printf("Server started\n");

while (1) {

read(pipe1[0], buffer, sizeof(buffer));

if (strcmp(buffer, "exit") == 0) break;

process\_data(buffer, response);

write(pipe2[1], response, strlen(response) + 1);

}

close(pipe1[0]);

close(pipe2[1]);

return 0;

}

void process\_data(const char\* input, char\* output) {

snprintf(output, 1024, "Processed: %s", input);

}

**Explanation:**

The server reads data sent by the client through pipe1.It processes the data using process\_data and sends the processed response back to the client through pipe2.

* It is recommended to use a starter (supervisor) process to launch client and server and to organize their communication

**Filename:** starter.c

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <string.h>

void start\_processes(int pipe1[2], int pipe2[2]);

int main() {

int pipe1[2], pipe2[2];

if (pipe(pipe1) == -1 || pipe(pipe2) == -1) {

perror("Pipe creation failed");

exit(1);

}

start\_processes(pipe1, pipe2);

return 0;

}

void start\_processes(int pipe1[2], int pipe2[2]) {

pid\_t pid\_client, pid\_server;

if ((pid\_client = fork()) == 0) {

close(pipe1[0]);

close(pipe2[1]);

execl("./client", "./client", NULL);

perror("Failed to start client");

exit(1);

}

if ((pid\_server = fork()) == 0) {

close(pipe1[1]);

close(pipe2[0]);

execl("./server", "./server", NULL);

perror("Failed to start server");

exit(1);

}

close(pipe1[0]);

close(pipe1[1]);

close(pipe2[0]);

close(pipe2[1]);

wait(NULL);

wait(NULL);

}

**Explanation:**

The supervisor creates two pipes for inter-process communication.It spawns two child processes: the client and the server.Communication between the client and the server is facilitated by the pipes.

* Debug and test the programs

gcc -o starter starter.c

gcc -o client client.c

gcc -o server server.c

./starter

5. IPC via messages (W9)

Exercise: Please implement the required information processing using a pair of specialized processes – client and server; provide the information communication between the processes via Linux messages.

* Use the data processing task of W1

**Filename:** mq\_common.c

#ifndef MQ\_COMMON\_H

#define MQ\_COMMON\_H

#define MQ\_CLIENT\_TO\_SERVER "/mq\_client\_to\_server"

#define MQ\_SERVER\_TO\_CLIENT "/mq\_server\_to\_client"

#define MAX\_MESSAGE\_SIZE 1024

#endif

**Explanation**:

Contains common definitions for message queues, such as names and maximum message size.

* Compose a client that periodically inputs data and sends it to the server

**Filename:** client\_mq.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <mqueue.h>

#include "mq\_common.h"

int main() {

mqd\_t mq\_client\_to\_server, mq\_server\_to\_client;

char input[MAX\_MESSAGE\_SIZE];

char response[MAX\_MESSAGE\_SIZE];

mq\_client\_to\_server = mq\_open(MQ\_CLIENT\_TO\_SERVER, O\_WRONLY);

mq\_server\_to\_client = mq\_open(MQ\_SERVER\_TO\_CLIENT, O\_RDONLY);

if (mq\_client\_to\_server == -1 || mq\_server\_to\_client == -1) {

perror("Failed to open message queues");

exit(EXIT\_FAILURE);

}

printf("Client started. Type 'exit' to quit.\n");

while (1) {

printf("Enter data to send to the server: ");

fgets(input, MAX\_MESSAGE\_SIZE, stdin);

input[strcspn(input, "\n")] = '\0';

if (strcmp(input, "exit") == 0) break;

if (mq\_send(mq\_client\_to\_server, input, strlen(input) + 1, 0) == -1) {

perror("Failed to send message");

break;

}

if (mq\_receive(mq\_server\_to\_client, response, MAX\_MESSAGE\_SIZE, NULL) == -1) {

perror("Failed to receive message");

break;

}

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

}

mq\_close(mq\_client\_to\_server);

mq\_close(mq\_server\_to\_client);

return 0;

}

**Explanation:**

Implements the client-side logic for sending data to the server and receiving the response using message queues.

* Compose a server that receives data from the client, processes it, and returns the result back to the client

**Filename:** server\_mq.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <mqueue.h>

#include "mq\_common.h"

void process\_data(const char \*input, char \*output) {

snprintf(output, MAX\_MESSAGE\_SIZE, "Processed: %s", input);

}

int main() {

mqd\_t mq\_client\_to\_server, mq\_server\_to\_client;

char buffer[MAX\_MESSAGE\_SIZE];

char response[MAX\_MESSAGE\_SIZE];

mq\_client\_to\_server = mq\_open(MQ\_CLIENT\_TO\_SERVER, O\_RDONLY | O\_CREAT, 0644, NULL);

mq\_server\_to\_client = mq\_open(MQ\_SERVER\_TO\_CLIENT, O\_WRONLY | O\_CREAT, 0644, NULL);

if (mq\_client\_to\_server == -1 || mq\_server\_to\_client == -1) {

perror("Failed to open message queues");

exit(EXIT\_FAILURE);

}

printf("Server started.\n");

while (1) {

if (mq\_receive(mq\_client\_to\_server, buffer, MAX\_MESSAGE\_SIZE, NULL) == -1) {

perror("Failed to receive message");

continue;

}

if (strcmp(buffer, "exit") == 0) break;

process\_data(buffer, response);

if (mq\_send(mq\_server\_to\_client, response, strlen(response) + 1, 0) == -1) {

perror("Failed to send message");

continue;

}

printf("Processed message: %s\n", response);

}

mq\_close(mq\_client\_to\_server);

mq\_close(mq\_server\_to\_client);

return 0;

}

**Explanation:**

Implements the server-side logic for receiving data from the client, processing it, and sending the processed response back.

* It is recommended to use a starter (supervisor) process to launch client and server and to organize their communication

**Filename:** starter\_mq.c

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

#include <mqueue.h>

#include "mq\_common.h"

void cleanup\_message\_queues() {

mq\_unlink(MQ\_CLIENT\_TO\_SERVER);

mq\_unlink(MQ\_SERVER\_TO\_CLIENT);

}

int main() {

pid\_t pid\_client, pid\_server;

cleanup\_message\_queues();

if ((pid\_client = fork()) == 0) {

execl("./client\_mq", "./client\_mq", NULL);

perror("Failed to start client");

exit(EXIT\_FAILURE);

}

if ((pid\_server = fork()) == 0) {

execl("./server\_mq", "./server\_mq", NULL);

perror("Failed to start server");

exit(EXIT\_FAILURE);

}

wait(NULL);

wait(NULL);

cleanup\_message\_queues();

printf("Client and server processes finished. Message queues cleaned up.\n");

return 0;

}

**Explanation:**

Supervises the client and server processes by forking and initializing the required resources (message queues).

* Debug and test the programs

gcc -o client\_mq client\_mq.c -lrt

gcc -o server\_mq server\_mq.c -lrt

gcc -o starter\_mq starter\_mq.c -lrt

./starter\_mq

6. IPC via semaphores and shared segments of memory (W10)

Exercise: Please implement the required information processing using a pair of specialized processes – client and server; provide the information communication between the processes via Linux semaphores and shared segments of memory.

* Use the data processing task of W1

**Filename:** shm\_common.h

#ifndef SHM\_COMMON\_H

#define SHM\_COMMON\_H

#define SHM\_KEY 1234

#define SEM\_KEY 5678

#define MAX\_MESSAGE\_SIZE 256

#endif

**Explanation:**

This header file defines shared constants such as keys for shared memory and semaphores, along with the maximum message size. It ensures synchronization and shared memory communication are standardized across the client, server, and supervisor processes.

* Compose a client that periodically inputs data and sends it to the server

**Filename:** client\_shm.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <sys/sem.h>

#include <unistd.h>

#include "shm\_common.h"

int main() {

int shm\_id, sem\_id;

char\* shared\_memory;

shm\_id = shmget(SHM\_KEY, MAX\_MESSAGE\_SIZE, 0666);

if (shm\_id == -1) {

perror("Failed to attach to shared memory");

exit(1);

}

shared\_memory = (char\*)shmat(shm\_id, NULL, 0);

sem\_id = semget(SEM\_KEY, 2, 0666);

if (sem\_id == -1) {

perror("Failed to attach to semaphores");

exit(1);

}

while (1) {

printf("Enter data to send to server (or 'exit' to quit): ");

fgets(shared\_memory, MAX\_MESSAGE\_SIZE, stdin);

shared\_memory[strcspn(shared\_memory, "\n")] = '\0';

if (strcmp(shared\_memory, "exit") == 0) {

signal\_semaphore(sem\_id, 1);

break;

}

signal\_semaphore(sem\_id, 1);

wait\_semaphore(sem\_id, 0);

printf("Server response: %s\n", shared\_memory);

}

shmdt(shared\_memory);

return 0;

}

**Explanation:**

The client attaches to the shared memory and semaphore. It reads user input and writes it to shared memory. Signals the server semaphore after sending a message and waits for the server's response. If "exit" is sent, the loop terminates.

* Compose a server that receives data from the client, processes it, and returns the result back to the client

**Filename:** server\_shm.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <sys/sem.h>

#include <unistd.h>

#include "shm\_common.h"

int main() {

int shm\_id, sem\_id;

char\* shared\_memory;

shm\_id = shmget(SHM\_KEY, MAX\_MESSAGE\_SIZE, 0666);

if (shm\_id == -1) {

perror("Failed to attach to shared memory");

exit(1);

}

shared\_memory = (char\*)shmat(shm\_id, NULL, 0);

sem\_id = semget(SEM\_KEY, 2, 0666);

if (sem\_id == -1) {

perror("Failed to attach to semaphores");

exit(1);

}

while (1) {

wait\_semaphore(sem\_id, 1);

if (strcmp(shared\_memory, "exit") == 0) {

break;

}

char processed[MAX\_MESSAGE\_SIZE];

snprintf(processed, MAX\_MESSAGE\_SIZE, "Processed: %s", shared\_memory);

strcpy(shared\_memory, processed);

signal\_semaphore(sem\_id, 0);

}

shmdt(shared\_memory);

return 0;

}

**Explanation:**

The server attaches to shared memory and semaphore. It waits for the client to signal, processes the data by appending "Processed: ", and writes the result back to shared memory. It signals the client semaphore to indicate a response is ready.

* It is recommended to use a starter (supervisor) process to launch client and server and to organize their communication

**Filename:** supervisor\_shm.c

#include <stdio.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <sys/sem.h>

#include <unistd.h>

#include <sys/wait.h>

#include "shm\_common.h"

void cleanup(int shm\_id, int sem\_id) {

shmctl(shm\_id, IPC\_RMID, NULL);

semctl(sem\_id, 0, IPC\_RMID);

}

int main() {

int shm\_id, sem\_id;

shm\_id = shmget(SHM\_KEY, MAX\_MESSAGE\_SIZE, IPC\_CREAT | 0666);

if (shm\_id == -1) {

perror("Failed to create shared memory");

exit(1);

}

sem\_id = semget(SEM\_KEY, 2, IPC\_CREAT | 0666);

if (sem\_id == -1) {

perror("Failed to create semaphores");

cleanup(shm\_id, sem\_id);

exit(1);

}

semctl(sem\_id, 0, SETVAL, 1);

semctl(sem\_id, 1, SETVAL, 0);

if (fork() == 0) {

execl("./client\_shm", "./client\_shm", NULL);

perror("Failed to launch client");

exit(1);

}

if (fork() == 0) {

execl("./server\_shm", "./server\_shm", NULL);

perror("Failed to launch server");

exit(1);

}

wait(NULL);

wait(NULL);

cleanup(shm\_id, sem\_id);

printf("Supervisor: Cleaned up resources.\n");

return 0;

}

**Explanation:**

This supervisor process creates the shared memory and semaphores, initializes the semaphores, and forks client and server processes. It cleans up resources after execution.

* Debug and test the programs

gcc -o supervisor\_shm supervisor\_shm.c

gcc -o client\_shm client\_shm.c

gcc -o server\_shm server\_shm.c

./supervisor\_shm

7. IPC via IP-UDP sockets (W11)

Exercise: Please implement the required information processing using a pair of specialized processes – client and server; provide the information communication between the processes via Linux IP-UDP sockets.

* Use the data processing task of W1

**Filename:** udp\_server.c

void process\_data(const char\* input, char\* output) {

snprintf(output, MAX\_MESSAGE\_SIZE, "Processed: %s", input);

}

**Explanation:**

This function is responsible for processing the input data received from the client. It formats the message by appending "Processed: " to the beginning of the input string.

* Compose a client that periodically inputs data and sends it to the server

**Filename:** udp\_client.c

int main() {

int sockfd;

struct sockaddr\_in server\_addr;

char buffer[MAX\_MESSAGE\_SIZE], response[MAX\_MESSAGE\_SIZE];

socklen\_t addr\_len = sizeof(server\_addr);

// Create a socket

if ((sockfd = socket(AF\_INET, SOCK\_DGRAM, 0)) < 0) {

perror("Failed to create socket");

exit(EXIT\_FAILURE);

}

server\_addr.sin\_family = AF\_INET;

server\_addr.sin\_port = htons(SERVER\_PORT);

if (inet\_pton(AF\_INET, SERVER\_IP, &server\_addr.sin\_addr) <= 0) {

perror("Invalid server IP address");

close(sockfd);

exit(EXIT\_FAILURE);

}

while (1) {

printf("Enter a message to send to the server (or 'exit' to quit): ");

fgets(buffer, MAX\_MESSAGE\_SIZE, stdin);

buffer[strcspn(buffer, "\n")] = '\0';

if (strcmp(buffer, "exit") == 0) break;

if (sendto(sockfd, buffer, strlen(buffer), 0, (struct sockaddr\*)&server\_addr, addr\_len) < 0) {

perror("Failed to send message");

continue;

}

int recv\_len = recvfrom(sockfd, response, MAX\_MESSAGE\_SIZE, 0, NULL, NULL);

if (recv\_len < 0) {

perror("Failed to receive response");

continue;

}

response[recv\_len] = '\0';

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

}

close(sockfd);

return 0;

}

**Explanation:**

The client periodically takes user input, sends it to the server using a UDP socket, and waits for a response. If the user types "exit," the client terminates.

* Compose a server that receives data from the client, processes it, and returns the result back to the client

**Filename:** udp\_server.c

int main() {

int sockfd;

struct sockaddr\_in server\_addr, client\_addr;

char buffer[MAX\_MESSAGE\_SIZE], response[MAX\_MESSAGE\_SIZE];

socklen\_t addr\_len = sizeof(client\_addr);

// Create a socket

if ((sockfd = socket(AF\_INET, SOCK\_DGRAM, 0)) < 0) {

perror("Failed to create socket");

exit(EXIT\_FAILURE);

}

server\_addr.sin\_family = AF\_INET;

server\_addr.sin\_addr.s\_addr = INADDR\_ANY;

server\_addr.sin\_port = htons(SERVER\_PORT);

if (bind(sockfd, (struct sockaddr\*)&server\_addr, sizeof(server\_addr)) < 0) {

perror("Bind failed");

close(sockfd);

exit(EXIT\_FAILURE);

}

printf("Server is running and waiting for messages...\n");

while (1) {

int recv\_len = recvfrom(sockfd, buffer, MAX\_MESSAGE\_SIZE, 0, (struct sockaddr\*)&client\_addr, &addr\_len);

if (recv\_len < 0) {

perror("Failed to receive data");

continue;

}

buffer[recv\_len] = '\0';

process\_data(buffer, response);

if (sendto(sockfd, response, strlen(response), 0, (struct sockaddr\*)&client\_addr, addr\_len) < 0) {

perror("Failed to send response");

}

printf("Processed message: '%s'\n", response);

}

close(sockfd);

return 0;

}

**Explanation:**

The server listens for incoming client messages using a UDP socket, processes them using the process\_data function, and sends the processed response back to the client.

* Use coordinated identification of client and server with a socket – IP address and port number.

**Filename:** udp\_client.c and udp\_server.c

Code Snippet (Client):

#define SERVER\_IP "127.0.0.1"

#define SERVER\_PORT 8080

Code Snippet (Server):

server\_addr.sin\_addr.s\_addr = INADDR\_ANY;

server\_addr.sin\_port = htons(SERVER\_PORT);

**Explanation:**

The client and server are coordinated by sharing the same IP address and port number. The client explicitly uses the server's IP address (127.0.0.1) and port (8080), while the server binds to the same port to accept client messages.

* Debug and test the programs

Compile the programs:

gcc -o udp\_server udp\_server.c

gcc -o udp\_client udp\_client.c

Run the server:

./udp\_server

Run the client:

./udp\_client