NETWORK PROGRAMMING MIDTERM HACKATHON SESSION 2

Sena Kılınç 210316036

Manisa Celal Bayar University Faculty of Engineering

Manisa, TURKEY

senakilinc2103@gmail.com

Abstract

In this project, problem-oriented; Updates of applications installed on the server are checked by multiple clients. If the application is up to date, no action is required, but if it is not up to date, a download link is given to the client. Message queue is used for messaging and shared memory is used for URL.

Methodology: For this purpose, two basic IPC (Inter-Process Communication) mechanisms were used: Message Queues and Shared Memory. The first piece of code involves the use of message queues for client-side operations, while the second piece of code involves server-side operations, i.e. responding to client requests and providing update URLs over shared memory as needed.

1. Introduction

1.1 Message queue

Message Queuing, in the context of inter-process communication (IPC), is a method used to allow processes to communicate with each other, typically in a system where these processes might be running at different times. A message queue is created or accessed using a unique key. Messages in a queue are typically structured. They often consist of a message type (or priority) and a text segment where the actual data is stored. Message queues provide a synchronized communication mechanism, meaning that the processes do not need to read or write messages at the same time. The queue safely holds the messages until they are processed.

Advantages:

Decoupling: The sender and receiver processes do not need to interact with each other directly. They just need to agree on the structure of the messages and the queue key.

Flexibility: Messages can be processed as needed without requiring both processes to be active and connected at the same time.

Scalability: New clients can be added easily, as they just need to use the same queue for communication.

1.2 Shared Memory

What a Shared Memory Works:

Using shared memory for sharing information like download URLs between a client and a server is an advanced technique in inter-process communication (IPC). Shared memory allows two or more processes to access a common memory space, which is faster than other forms of IPC because data does not need to be copied between the client and server. A segment of memory is allocated for shared access. In Unix-like systems, this is typically done using *shmget*, which creates a new shared memory segment, or accesses an existing one.

Processes that wish to use this shared memory must attach themselves to it. This is done using *shmat* in Unix-like systems. After

attachment, the process can access the shared memory segment as if it were a part of its own address space.

Once attached, processes can read from and write to the shared memory. In your case, the server can write a download URL into the shared memory, and the client can read this URL.

When the processes have finished using the shared memory, they should detach from it using *shmdt*. When it's no longer needed at all, the shared memory segment can be removed with *shmctl*.

Client-Server Communication Using Shared Memory:

When the server decides that a client needs an update, it sets up a shared memory segment and writes the download URL to this segment.

The server then sends a message to the client (possibly via a message queue) containing the identifier (key or ID) of the shared memory segment.

Upon receiving this message, the client uses the provided key or ID to attach to the shared memory segment. It then reads the download URL.

After reading the URL, the client detaches from the shared memory segment. The client can then use the URL to download the necessary update.

Once the server knows that the client has retrieved the URL (which may involve additional messaging or checks), it can remove the shared memory segment.

1.3 Libraries used in this problem

#include <stdio.h>: Includes functions for standard input/output operations.
#include <stdlib.h>: Contains functions for memory management, random number generation, conversion operations.
#include <sys/ipc.h>: Contains key management functions for IPC.
#include <sys/msg.h>: Contains structures and functions related to the message queue.
#include <sys/shm.h>: Contains structures and functions for shared memory operations.
#include <string.h>: Includes functions for string operations.
#include <time.h>: Includes functions for time-related operations.

2. My Approach

2.1 Client Side

This code allows the client to interact with the server and perform update checking using IPC mechanisms. When an update is required, the processes of obtaining a URL from shared memory and using this URL are coded. This method allows multiple clients to interact with the same server and check for updates.

Purpose and Functionality

This piece of code allows a client to send a query to a server to check whether the application it has installed is up to date.

Message Queue Structure Description:
struct *msg_buffer*: Defines a structure for the message queue. *msg_type* is a long integer type and *msg_text* can contain messages of up to 100 characters.

Main Function Start:
int main(): Starts the main function of the program.

Program Information Description:
char programID[ ] = "0000001 V2.3 CinsCalculator";: Stores the ID, version and name of the program as a string.

Creating a Message Queue Key:
key_t key = ftok("/home", 'M');: Creates a unique IPC key using the /home path and the 'M' character.

Message Queue Creation/Control:
int msgid = msgget(key, 0666 | IPC_CREAT);: Creates a message queue with the specified key or accesses an existing queue. If there is no queue, it creates a new queue.

Message Queuing Error Checking:
if (msgid == -1) {: Returns -1 if *msgget* function fails. In this case, an error message is printed, and the program is terminated.

Message Preparation:

message.msg_type = 1;: Sets the message type to 1.
strcpy(message.msg_text, programID);: Copies the program ID information to the text part of the message.

Do not sent me a message:
if (msgsnd(msgid, &message, sizeof(message.msg_text), 0) == -1) {: Sends the prepared message to the message queue. If it fails, it prints an error message.

Getting Response from Server:
if (msgrcv(msgid, &message, sizeof(message.msg_text), 1, 0) == -1) {: Retrieves the response from the server from the message queue. If it fails, it prints an error message.

Checking and Processing Update Status:
if (strcmp(message.msg_text, token) == 0) {: Compares the server's response with the current version of the program. If it matches, it indicates that the program is up to date. If an update is required, it accesses and retrieves the shared memory segment containing the update URL.

Allocating and Deleting Shared Memory:
shmdt(shmURL);: Allocates shared memory.
shmctl(shmid, IPC_RMID, NULL);: Deletes the shared memory segment.

Updated Program Information:
strncpy(programID, message.msg_text, ...);: Updates the new program information received from the server with the current program ID.

End of Main Function:
return 0;: Indicates that the main function has been completed successfully and terminates the program.

2.2 Server Side

Purpose and Functionality

This piece of code acts as a server that processes update queries from clients and provides update information if necessary.

Message Queue Structure Description:
struct msg_buffer: Defines a structure for the message queue. msg_type is a long integer, and msg_text is a text field 100 characters long.

Main Function and Variable Definitions:
int main(): Starts the main function of the program.
char newVersions[6]*: An array used to store new program versions.
char newURLs[6]:* An array used to store download URLs of new versions.

Message Queue and Shared Memory Keys:
ftok: Generates unique IPC keys using specified paths and characters.

Message Queue Creation/Control:
msgget: Creates a message queue with the specified key or accesses an existing queue.

Message Queuing Error Checking:
In case of error, it prints the appropriate message and exits the program.

Server Cycle:
while (1): It is a loop that runs continuously. The server continues to receive messages from clients during this cycle.

Receiving Message from Client:
msgrcv: Retrieves messages from clients from the message queue.

Message Processing and Reply Sending:
It checks whether the program is up to date based on the message content.
If it is not up to date, it creates shared memory and writes the update information there.

Shared Memory Operations:
shmget: Creates a shared memory segment.
shmat: Provides access to the created shared memory segment.

Sending Response Message to Client:
msgsnd: Sends a message in response to the client.

End of Program:
return 0;: Indicates that the main function has been completed successfully and the program has been terminated.

This code allows a server to process update requests from multiple clients, share the URL over shared memory if an update is required, and send this information back to the clients. Message queuing and shared

memory usage are commonly used methods for effective communication and data sharing in such distributed systems.

2.3 Integration and Collaboration

These two pieces of code work together to allow multiple clients to check whether the applications they load from the server are up to date and update them if necessary. The client code sends a query to the server and the server returns the appropriate response based on the update status. This is accomplished using network messaging and shared memory mechanisms.

2.4 Conclusion and Evaluation

These codes meet the requirement of multiple clients to check for application updates from the server. However, they need further development for use in real-world scenarios. Additions and improvements may be required, especially in security, data accuracy and communication over the network.

3. Alternatives

Message queues and shared memory can be replaced with more contemporary and safe options like RESTful APIs and web services. Secure network data transport is made possible by these techniques.
Update distribution and administration can be streamlined and centralized with the use of cloud-based technologies.

4. Fixes and Improvements

To improve security, the system should be outfitted with techniques like encryption and permission.
One important area where the system has to be improved is error management and resilience to network disruptions.

5. Conclusion with Recommendations for the Future

a) Security and Reliability Focused Approach: When it comes to updating mechanisms, security and dependability should come first. Secure authentication techniques and encryption ought to be essential components of these procedures.

b) Transition to Modern Technologies: These technologies can be replaced with more controllable and secure options thanks to RESTful APIs and cloud-based services.

c) Flexibility and Scalability: The system needs to be flexible and scalable in order to function well under a range of user scenarios and loads.

d) Continuous Improvement: The system needs to be thoroughly tested and updated on a regular basis. Early vulnerability and error detection is made possible by this method.

As a result, this system meets the need for multiple clients to check for application updates from the server; However, it needs to be constantly improved and adapted to modern technologies.

Additional Code

1. Client Side

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/shm.h>
#include <string.h>
#include <time.h>
// Structure for message queue
struct msg_buffer {
    long msg_type;
    char msg_text[100];
} message;
int main() {
    // Program's ID, version, and name
    char programID[] = "0000001 V2.3 CinsCalculator";
    // Creating a key for the message queue
    key_t key = ftok("/home", 'M');
    // Attempt to create or access a message queue
    int msgid = msgget(key, 0666 | IPC_CREAT);
    if (msgid == -1) {
        perror("Message queue creation failed");
        exit(EXIT_FAILURE);}
    // Preparing the message to be sent
    message.msg_type = 1;
    strcpy(message.msg_text, programID);
```

```c
    // Sending the message
    if (msgsnd(msgid, &message, sizeof(message.msg_text), 0) == -1) {
        perror("Failed to send message");
    } else {
        // Receiving the response from the server
        if (msgrcv(msgid, &message, sizeof(message.msg_text), 1, 0) == -1) {
            perror("Failed to receive message");
        } else {
            char *token = strtok(programID, " ");
            if (strcmp(message.msg_text, token) == 0) {
                printf("Program is up to date\n");
            } else {
                // Processing the update message
                int shmid = atoi(strrchr(message.msg_text, ' ') + 1);
                char *shmURL = (char *)shmat(shmid, NULL, 0);
                printf("URL is %s\n", shmURL);
                printf("Program is updated at %s\n", asctime(localtime(&(time_t){time(NULL)})));
                // Detaching and removing shared memory
                shmdt(shmURL);
                shmctl(shmid, IPC_RMID, NULL);
                strncpy(programID, message.msg_text, strrchr(message.msg_text, ' ') - message.msg_text);
                printf("New programID: %s\n", programID);
            }
        }
    }
    return 0;// Continue with the rest of the program
}
```

2. Server Side

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/shm.h>
#include <string.h>
// Define structure for message queue
struct msg_buffer {
    long msg_type;
    char msg_text[100];
} receivedMsg, sendedMsg;
int main() {
    // Array of new program versions
    char* newVersions[6]; // Initialize with actual versions
    // Array of URLs corresponding to new versions
    char* newURLs[6]; // Initialize with actual URLs

    // Creating keys and setting up message queue
    key_t msgKey = ftok("/home", 'M');
    int msgid = msgget(msgKey, 0666 | IPC_CREAT);
    if (msgid == -1) {
        perror("Failed to create/access message queue");
        exit(EXIT_FAILURE);
    }
    // Creating key for shared memory
    key_t shmKey = ftok("/temp", 'C');
    int shmID;
    // Server's ID for message queue operations
    int serverID = 0;S
```

```c
    // Server loop
    while (1) {
        // Receiving messages from clients
        if (msgrcv(msgid, &receivedMsg, sizeof(receivedMsg.msg_text), serverID, 0) == -1) {
            perror("Error receiving message");
            exit(EXIT_FAILURE);
        } else {
            // Check received message against new versions
            for (int i = 0; i < sizeof(newVersions) / sizeof(newVersions[0]); i++) {
                // Extract program ID from message
                char* programID = strtok(receivedMsg.msg_text, " ");
                char* versionID = strtok(NULL, " ");

                // Check if the program ID matches and requires an update
                if (strcmp(programID, newVersions[i]) == 0) {
                    if (strcmp(receivedMsg.msg_text, newVersions[i]) == 0) {
                        // Send back the same message if up to date
                        strcpy(sendedMsg.msg_text, newVersions[i]);
                        sendedMsg.msg_type = strtol(programID, NULL, 10);
                        msgsnd(msgid, &sendedMsg, sizeof(sendedMsg.msg_text), 0);
                    } else {
                        // Create shared memory and send updated info
                        shmID = shmget(shmKey, 30, IPC_CREAT | 0666);
                        char* shared_data = (char*)shmat(shmID, NULL, 0);
                        strcpy(shared_data, newURLs[i]);

                        sprintf(sendedMsg.msg_text, "%s %d", newVersions[i], shmID);
                        sendedMsg.msg_type = strtol(programID, NULL, 10);
                        msgsnd(msgid, &sendedMsg, sizeof(sendedMsg.msg_text), 0);
                    }
                    break;
                }
            }
        }
    }
    return 0;
}
```