UNIVERSITY OF TRENTO

Department of Information Engineering and Computer Science

Dipartimento di Ingegneria e Scienza dell'Informazione:

Università degli studi di Trento

Anno formativo 2022/2023

Corso di laurea in Scienze e Tecnologie Informatiche

Sistemi Operativi

Titolo

Project :
“IPC-SO”

Degli allievi

Alex Cattoni, Dennis Cattoni,
Manuel Vettori

INDEX

1. Introduction

This project has the purpose of creating a dedicated kernel module, known as *IPC-SO*, which facilitates inter-process communication (IPC) without relying on existing techniques. The IPC-SO module has been designed to enable processes to engage in message-based communication, offering both synchronous and asynchronous modes of interaction.

Below are the constraints required for the development of the project:

1. Any process can make use of the new kernel module to send messages to other interested processes. The messages must be delivered in the order of their arrival.
2. Any process can receive messages in synchronous mode, where the process waits until a message arrives.
3. A 10-level priority system is implemented to allow processes to specify the importance of the messages they send. Messages with higher priority must be delivered before messages with lower priority.
4. A single message can be sent to multiple processes.
5. A message can be sent with any delay specified by the sender.
6. The received messages must contain: (i) any payload, (ii) the sender of the message, (iii) the priority of the message, and (iv) the potential delivery delay.
7. A message can be sent to processes utilizing IPC-SO, as well as to all members of a group.
8. IPC-SO must be impartial and fair, preventing the monopolization of communication by a single process.
9. The highest message priority (10) can be utilized by messages sent exclusively by a process with root privileges.
10. Messages can be received in asynchronous mode, with a mechanism that is as flexible and customizable as possible.

We found a compromise that meets the required constraints while also being efficient and flexible, while still taking into account the limitations of our architecture.

The details of the design choices will be explained later in this document in their respective chapters.
To generate the code documentation, we have chosen to use the *Doxygen* software, which allows the generation of documentation in HTML page format and LaTeX code.

In the respective *GitHub repository*, the code is provided along with a tutorial for setting up the environment, compiling, and launching.
Moreover, it explains how to perform code testing through the respective interface.

In the repository, there will also be a section dedicated to the primary resources we utilized to acquire the knowledge necessary for developing the project and a link to the documentation of the Kernel APIs.

It's important to note that the module has been developed for Ubuntu: 20.04.6 LTS (Focal Fossa). Therefore, if a different version of the operating system is used, we cannot guarantee the proper functioning of the module.

2. IPC-SO Architecture

2.1 General Idea

The general idea behind the Ipc mechanism is to use the respective custom device files to achieve communication between user processes and the kernel module.

Each device provides an interface that allows executing specific tasks defined by us:

- *Registration device*:
  This device has the purpose of enabling processes to register or unregister with the IPC-SO module.

- *Shared memory device*:
  This device allows the exchange of messages between processes and the module, enabling both message transmission from processes to the module and message retrieval from the module to user processes.

- *Synchronous device*:
  This device implements the logic for the synchronous communication mechanism.

2.2 Character Device File

Communication between User processes and the IPC-SO module occurs through three character devices named respectively: /registration, /shared_memory and /synchronous.

Character devices provide an interface for managing data at a character level. User-space processes can interact with these devices by performing read and write operations on the associated device file.

Common operations that can be performed on a character device file include:

Opening and Closing:

Character device files can be opened and closed like regular files. Opening the device file typically involves setting up the necessary data structures to facilitate communication with the device, while closing it involves cleaning up these structures.

Reading:

User programs can read data from the device file, which represents the data generated by the character-oriented device. Reading from the file descriptor associated with the character device triggers the retrieval of data from the device.

Writing:

User programs can write data to the device file, which sends the data to the character-oriented device for processing or output. Writing to the file descriptor associated with the character device allows sending data to the device.

The design of our module enables the proper handling of these operations, offering user processes a communication mechanism based on *message exchange*.

Each device has its own specific operations that are managed within the kernel module through the corresponding “fops” associated with any device.

All data sent to the kernel through the different device files is stored in respective queues of different types. This approach is chosen to provide significant flexibility to the module and ensure efficient management.

Each utilized queue has a *first* and a *last pointer*. This design enhances the efficiency of inserting new messages, as it avoids the necessity to traverse the entire queue every time.

2.3 Module initialization

The function “*ipc_os_module_init*” it’s an initialization routine for our module. The function is responsible for initializing all the utilities needed to interact correctly with the registration, shared_memory and synchronous device.

- *Shared Memory Device*:

This module starts by allocating a character device region dedicated to shared memory. A character device is then initialized using the corresponding file operations structure “fops_sm”.
Additionally, memory is allocated for the shared buffer ”shared_buffer” of a size of 4096 bytes, establishing the memory space for data sharing. If any step in this process fails, the function ensures proper cleanup and returns an appropriate error code.

- *Registration Device*:

Similar to the shared memory setup, a character device region is allocated for registration. The character device is initialized using the appropriate file operations structure “fops”. Memory allocation for the registration buffer “registration_buffer” of size 1024 bytes. In case of any failure during these steps, the function handles resource deallocation and returns the relevant error code.

- *Synchronous Memory Device*:

The same operations to initialize the shared memory and registration setups occur also for the synchronous. A character device region is allocated for synchronous memory. The character device is initialized with the appropriate file operations structure “fops_sync”. Memory allocation for the synchronous buffer “synchronous_buffer” is carried out of size 1024 bytes. In the case of an error in any stage, the function ensures cleanup and returns the corresponding error code.

2.4 Module deinitialization

The “*ipc_os_module_exit*” function handles the cleanup process when the IPC-SO module is unloaded from the kernel.

It releases resources associated with registration, priority queue, and synchronous memory. Memory allocated for shared and synchronous buffers is deallocated.

Device classes and nodes for shared memory, registration, synchronous memory are deallocated and corresponding character devices are unregistered.
Finally, an informational log message confirms the successful unloading of the IPC-OS module in the kernel logs.

2.5 Makefile

For enhanced flexibility during the compilation phase and ease of use, we have created two dedicated Makefiles: one for the *kernel module* and one for the *user interface*.

The kernel module's makefile offers the following commands:

- all:

  This command is executed with 'make' and facilitates the compilation of the module, generating the file IPC-SO.ko, which is then ready to be loaded into the system.

- clean:

  This command clears the kernel logs and removes all files in the directory except for "ipc-so.c," "IPC_SO.h," and the Makefile itself.

- insert:

  The insert command loads the module into the system and grants read and write permissions to the "*registration*," "*shared_memory*," and "*synchronous*" devices. This operation is crucial to establish the communication mechanism.

- list:

  This command prints the list of kernel logs to the terminal. It's useful for debugging and for identifying any potential errors.

- delete:

  The "delete" command allows the module to be properly unloaded from the system.

The user-specific Makefile offers the following commands:

- all:

  This command is launched with ‘make’ and it allows the user to compile the user_interface.c and generates the executable program called process_user.out.

- launch:

  This command allows the user to execute the user_interface.out without root permissions.

- root:

  The root command allows the user to execute the user_interface.out with root permissions

- clean:

  This command removes all files in the directory except for "user_process.c," "USER_PROCESS.h," and the Makefile itself.

3. User Interface

3.1 Interface

Our design provides a system based on message exchange, in both synchronous and asynchronous modes.

To interact with the IPC-SO module, we have created a dedicated interface that guides the user through the operations allowed by the module's architecture.
Before using the module, the first operation needed is the registration of the processes by their PIDs. This is done through the registration device.

After registration, you can perform one of the following operations:

- Unregister
- List of processes available
- Write
- Asynchronous Read
- Synchronous Read
- Exit

At any moment, the user can choose which operation to perform through the interface provided by the user_process.c file. The user can run the user_process.c an arbitrary number of times to have more available processes and test communication among them.

By selecting the *List of available processes*, the user will be provided with a list of processes registered with the IPC-SO module. This enables the user to easily choose the message recipient.

The *write* operation enables the user to compose a message, which can be accomplished in two ways:

- User Input: The interface prompts the user to populate the respective fields of the message using keyboard input.

- Auto: In this mode, the message is generated almost entirely automatically and randomly. The user will only be asked to input the recipient's PID. (Useful for debug and testing purposes)

The user can switch between these two write modes by setting the #define AUTO to 0 or 1 in the USER_PROCESS.h file.

In *Synchronous Read* mode, a process waits for the arrival of a message intended for it, preventing any other operations from taking place during the waiting period.
The process will only be woken up when a message intended for it arrives, allowing the user to choose the next operation.

In *Asynchronous Read* mode, a process requests to read all the messages destined for itself, including only the messages with the delay expired at the time of the read request. This reading mode doesn't block the process, and it can be performed by the user at any time.

The *Unsubscribe* operation allows the user to properly unsubscribe from the IPC-SO module. If the process is terminated abruptly or improperly, its PID will remain in the module, potentially causing confusion for the user. To prevent this, ensure that you unsubscribe your process properly.

The exit operation provides the user with a method to terminate the execution of user_process.c correctly.

3.2 Messages

The messages that our system enables to exchange have the following format:

```
//Type definition
typedef struct Message{
    int pid_recipient;    //recipient's PID
    int pid_sender;       //sender's PID
    int priority;         //priority of the message
    int delay;            //delay of the message
    char payload[128];    //payload in bytes
} Message;
```

The field ***pid_recipient*** represents the process that will receive the message, the field ***pid_sender*** indicates the PID of the process sending the message.

The ***priority*** field indicates the priority associated with the message, while the ***delay*** field indicates the delay in message delivery.

- ***NOTE***:

  Priority level 10 is only allowed for processes with *Root* permissions.
  If *user_process.c* is executed without sudo permissions (sudo ./user_process.) and the user attempts to assign a priority of 10 to a message, an "***Access denied***" error will occur.

  This situation can also arise in *#define AUTO 1* mode, as the random function might assign priority level 10 to a message from a process that wasn't launched with sudo permissions.

  If you encounter the "***Access denied***" error during the testing phase, it's due to the specific reason outlined in the IPC-SO requirements.

The ***payload*** field is dedicated to containing the message that the user wants to send. This field is limited to a length of 128 characters because it holds the byte conversion of an arbitrary payload. This implies that the maximum useful length for the payload is 64 alphanumeric symbols.

The decision to have a payload size of 64 was made as a compromise to achieve greater scalability in the quantity of messages within the kernel module and efficiency despite a fixed size.

A larger payload size could be chosen, but this would potentially result in a decrease in the efficiency of the mechanism and limited traffic.

4. Communication mechanism

To provide maximum flexibility to the system, we opted to use dedicated data types and structures.

For more implementation details, utilized functions, and optimizations, refer to the ***Doxygen*** documentation.

The primary data structure used is the queue, optimized to avoid full-length swiping during each insertion of a new item.
The core data structure, enabling delay and priority management, is an array containing queues for each priority level. Its name is “*Queue_mes **PQ*”.

4.1 Registration

Registration is the first action that a process must undertake to effectively utilize the IPC-SO mechanism.

The kernel module enables user processes to interact with it and register themselves.
Its primary purpose is to facilitate the registration of user processes within the kernel space, allowing them to participate in the inter-process communication system managed by the module.

When a user process interacts with the /dev/registration device file, it essentially communicates with the kernel module, indicating its intention to join the IPC-SO communication system. This interaction involves sending a process ID (PID) as a registration request. PIDs are unique identifiers assigned to each running process in the system.

Upon receiving a registration request, the kernel module employs a data structure known as a "*registration queue*." This queue is implemented as a linked list and serves as a temporary repository for the received PIDs.

The registration queue efficiently manages the PIDs of user processes that have registered with the module, and it also handles requests from user processes to view all the processes subscribed to IPC-SO.

4.2 Write

When a Message is generated from the interface provided by the "*user_process.c*" program, it is sent to the kernel module through the write() function, specifying the *shared_memory* device.

When a user process invokes the *write()* system call on the *shared_memory* device file, the *shared_memory_write()* function is invoked within the kernel module. Its main purpose is to receive the message from the user process and handle it appropriately as follows:

- Receiving and Parsing the Message:

  The data received from the user includes the message payload, priority level, recipient PID, and delay. These parameters define both the content of the message and its priority.

- Message Categorization:

  The received message is then categorized based on its priority level. The priority levels determine the urgency of the message within the IPC system. The function calculates the corresponding priority queue index based on the message's assigned priority minus 1 (to be compatible with the indices of the PQ struct) .

- Priority Queue enqueue:

  The message is then enqueued into the appropriate priority queue. The priority queues are implemented as multiple queues, each corresponding to a specific priority level. The function adds the message node to the tail of the corresponding priority queue and sets the *arrival time* of the current message.

  A global pointer named "*last*" stores the address of the *next field of the last node in each priority queue. This offers efficiency in insertion by eliminating the need for traversing the entire queue.
  This process ensures that messages are ordered based on their priority.

- Delay Handling:

  For messages that have a delay associated with them, the function considers their delay period. It calculates the time that has passed since the message was received and delays the message’s entry into the priority queue until the delay period has elapsed.
  This is necessary for both synchronous and asynchronous communication.

- Synchronization and Communication:

  Messages in the priority queues are ready for inter-process communication. They can be dequeued and processed by recipient processes based on the priority-based scheduling of the IPC system.

By managing the messages sent by user processes and categorizing them into priority queues, the shared_memory_write() function allows communication and synchronization between processes of messages with different priorities.

This approach ensures that messages are delivered in a timely manner based on their urgency, as requested by the requirements of the IPC-SO module.

4.3 Asynchronous communication

Processes have to be able to perform both *asynchronous* and *synchronous* communication.

The idea behind asynchronous communication is to enable the user to read all the messages intended for it, with the delay consumed, and then display them to the console at any time.

The terminal printing was implemented to ***demonstrate*** the proper functioning of communication. With appropriate modifications to the *user_process.c* source code, the user can save received messages and perform other custom operations.

The shared_memory_read() function is responsible for responding to read requests from user processes and managing the retrieval and delivery of messages stored within the IPC system's priority queues.

When a user process initiates the read() system call on the shared memory device file, the shared_memory_read() function is invoked within the kernel module. Its main purpose is to provide a mechanism for user processes to retrieve messages that are stored in the IPC system and ready for delivery.

Here's an overview of how the *shared_memory_read()* function operates:

- **Message Retrieval**:

  The function starts by scanning through the priority queues, checking for messages that are ready for delivery. This operation focuses on messages for which the associated delay period has expired.

- **Priority Queue Dequeue**:

  For each priority queue, the function dequeues messages that are ready to be delivered. These messages are then enqueued into a *buffer queue*, which is structured to temporarily store messages for efficient delivery.

  The dequeue from the PQ array is performed in a way that takes into account cases where a message can be saved, adjusting the pointers of the list to maintain the correct functioning of the structure.

  Here are the steps for different cases when performing a dequeue operation from the PQ array: (i refers to the i-th priority queue)

- *Case where the Message is the first and only in PQ[i]*:

  Adjust the pointer PQ[i] to NULL.
  Set the global *last_ptr[i] to NULL.

- Case where the Message is the first, but there are other messages in its PQ[i]:

  Set the PQ[i] pointer to the second message in the queue.

- Case where the Message is in the middle of PQ[i]:

  Set the previous pointer to the next message in line.

- Case where the Message is the last in PQ[i]:

  Set the global *last_ptr[i] to the previous message in the queue.

- **Buffer Queue Priority Enqueue**:

  The priorityDequeue() function from the PQ array copies the addresses of the target messages into the buffer queue. This specific queue is used to temporarily store multiple messages that are ready to be written to the user, and it is defined as follows:

```
typedef struct Queue_buffer{
    Message *ptr;               //buffer queue item
    struct Queue_buffer *next;  //pointer to the next element
} Queue_buffer;
```

  As you can see from the type definition, the field Message *ptr contains a pointer to a struct Message. This is done in order to copy the addresses of target messages from the PQ array to the buffer queue.

  The priorityDequeue() function adjusts the pointers in the PQ array (specifically in PQ[i]) when a target message is found. It then passes the addresses of these messages to the buffer queue for enqueueing.

  Within the buffer queue, the messages are further organized based on their original priority levels. This organization ensures that messages are delivered to the user process in a prioritized manner and timing.

- **Message Delivery**:

  Once the messages are organized within the buffer queue as described earlier, the function proceeds to dequeue messages from the buffer queue and deliver them to the user process using the read() call. The messages are sent sequentially, following the priority-based ordering.

  After delivering all relevant messages to the user process, the buffer queue is cleared, and the function is ready to handle additional read requests.

By implementing this approach, the shared_memory_read() function manages the process of retrieving and delivering messages to user processes. It ensures that messages are presented to the requesting process in the order of their priority for delivery. This mechanism is our key design for timely inter-process asynchronous communication offered by the IPC-SO module.

4.4 Synchronous communication

The concept behind synchronous communication is to allow the user to put a process into a sleep state, where it remains until a message intended for it triggers its reawakening.

Likewise the Asynchronous communication, the terminal printing was implemented to *demonstrate* the proper functioning of the communication mechanism. With appropriate modifications to the *user_process.c* source code, the user can save received messages and perform other custom operations.

The synchronous_read() function is responsible for responding to read requests from user processes, managing the retrieval and delivery of the message stored within the IPC system's priority queues and the synchronization between the processes sleeping.
When a user process initiates the read() system call on the synchronous device file, the synchronous_read() function is invoked within the kernel module. Its main purpose is to provide a mechanism for user processes to retrieve the first message available in the IPC system and ready for delivery.

Here's an overview of how the *synchronous_write()* functions operate:

When invoking the synchronous_write() function, the user's PID is stored within a queue called sync_pid. This queue keeps track of the PIDs of processes that are requesting synchronous reading.

Here's an overview of how the *synchronous_read()* function operates:

- **Sync_PID Retrieval**:

  The function starts by scanning through the sync_queue, checking if the PID of the process calling is contained into it. If found, it returns the sync_pid struct associated with the corresponding PID.

- **Finding message in the PQ[]**:

  Right after checking if the PID is part of the sync_queue I check if there's a message into the PQ by calling the function *sync_mes_check_find()*, two cases appear:
  - a message destined for the calling process exists (Case 1)
  - a message destined for the calling process doesn't exist (Case 2)

- **Case 1: Message is in the PQ**

  In this scenario, the system fetches the message, accurately computes the message's actual delay, and puts the process to sleep for the intended duration (in case of more messages, it takes the minimal actual delay).
  Determining the actual delay of the message involves a comparison between the arrival times of the reading request of the process and the message, as well as an assessment of the message delay.
  The message's information is then stored in the sync_pid structure (delay, arrival, actual_delay).

- **Case 2: Message is not in the PQ**

  In this scenario, the process is put to sleep, awaiting a trigger. It will be reactivated upon a request to the shared_memory_write() function, provided a message has been directed towards it.
  Subsequent to reawakening, the process will once more enter a sleeping state, this time aligning its sleep duration with the delay associated with the incoming message.
  The message's information is then stored in the sync_pid structure (delay, arrival, actual_delay).

- **A new message with less delay is sent**

  In this situation, it compares the actual delay of the message already present within the sync_pid structure in the sync_queue with the delay of the incoming message. If the new delay is shorter, it adjusts the process's wake-up delay and updates the information within its sync_pid structure. Otherwise, if the delay is longer no alterations will occur.

- **Message Delivery**:

  Once the process's timer has completed its countdown the function proceeds to dequeue the message and deliver it to the user process using the read() call and the process wakes up.

By implementing this approach, the synchronous_read() function manages the process of retrieving and delivering a message to the user process waiting for it.
This mechanism is our key design for timely inter-process synchronous communication offered by the IPC-SO module.

5. Documentation and Github

5.1 Doxygen doc

For the technical documentation of the source code, we chose to use the ***Doxygen*** software. The documentation provides technical and implementation details about the data types, functions, parameters and return values.

The generated documentation is available in two formats: LaTeX code and static HTML pages.
The LaTeX code can be then used to generate the documentation in PDF format.
Instead the static HTML pages provide a user-friendly format that is easily accessible through web browsers. This makes it simple to navigate and understand the source code.

If the source code is modified, it's necessary to properly insert Doxygen comments and then compile to obtain the updated documentation.
This approach ensures good maintainability of the code and maintains consistency and accuracy in documentation across future developments of the project.

5.2 Github

The GitHub repository contains the IPC-SO module project, you can find instructions for setting up the environment to properly use the module.

Additionally, there will be a dedicated section called "How to Use" that explains how to utilize the kernel module effectively.
It also includes a "how to use" guide, references to the most useful materials used during the project development, and credits to the developers involved.

Link to the repository:

https://github.com/denniscattoni/IPC-SO

6. Constraints and limitations

We are aware of the limitations that the IPC-SO module presents and the potential optimizations that can be made in future modifications. We outline the limitations resulting from the design of the module:

Limited Message Size:
The fixed size of the payload restricts the amount of data that can be transmitted within each message. This limitation might not work properly in scenarios where larger chunks of data need to be exchanged between processes.

Limited device buffer:
Each device has a fixed-size buffer. This limitation implies that the amount of data that can be stored in these buffers is restricted to their predefined sizes. If the data being processed or communicated exceeds these sizes, it could result in data loss or truncation.

Memory Usage:
The architecture allocates memory for various data structures, including queues and message buffers. With a large number of processes and messages, memory usage can become a concern, potentially leading to memory exhaustion or inefficient memory management.

Handshake of Message Delivery:
The current architecture doesn’t provide mechanisms for guaranteeing message delivery. While it handles message queues and priorities, there's no acknowledgment or confirmation mechanism to ensure that messages are successfully received by the intended recipients.

Usability:
The architecture enforces a specific interface and usage pattern for interaction with the IPC system. This lack of flexibility might make it challenging to adapt the system to different communication needs or to integrate it with other systems.

Scalability:
While the architecture supports prioritization of messages, it might not scale well to a large number of messages or processes. Managing and scheduling messages efficiently as the system grows could become a challenge.

7. Tasks achieved

As specified in the introduction, considering the constraints required for the development of IPC-SO, we have completed all the requested points except for requirement ***number 7***.

This decision was primarily made to avoid further complicating the code and to focus on optimization, debugging, and creating more accurate and precise documentation.

One of the major challenges we found was in the implementation of the delay feature, which significantly increased the complexity of the synchronous Read operation.

In future developments, we believe that the mechanism could be made more flexible, potentially by removing the constraint of the maximum payload size through more sophisticated mechanisms.

8. Bibliography

8.1 References

To develop this project, we had to study new concepts and delve deeper into many Kernel programming notions.

The following guide was of fundamental importance for deepening some concepts necessary for the development of the project and to design the architecture:

- The Linux Kernel Module Programming Guide: https://sysprog21.github.io/lkmpg/

To gather more information about primitives and APIs, we referred to the official documentation available here:

- Kernel API: https://www.kernel.org/doc/html/latest/core-api/index.html
- Waiting Queues: https://t.ly/d4L48

This playlist was useful to show a trivial example of a Kernel module:

- Kernel Module Development Playlist: https://rb.gy/tv2cj

8.1 Credits

- *Alex Cattoni* - Università degli studi di Trento (Unitn), Trento – Italy
  alex.cattoni@studenti.unitn.it

- *Dennis Cattoni* - Università degli studi di Trento (Unitn), Trento – Italy
  dennis.cattoni@studenti.unitn.it

- *Manuel Vettori* - Università degli studi di Trento (Unitn), Trento – Italy
  manuel.vettori@studenti.unitn.it
