Inter-Process Communication (IPC) in Novix OS

In a multitasking operating system, processes often need to **communicate** and **cooperate**. Whether it's a shell sending a command to a graphical server or background tasks sharing data, **Inter-Process Communication (IPC)** is the bridge that makes this possible.

In this chapter, we'll design and implement a simple but powerful **IPC mechanism** for Novix OS, allowing processes to exchange messages safely through the kernel.

Learning Goals

By the end of this chapter, you'll understand:

- What IPC is and why it's essential for multitasking systems
- How Novix OS implements message-based IPC
- How `ipc_send()` and `ipc_recv()` work internally
- How a shell process can send commands to a window server

What is IPC?

Inter-Process Communication (IPC) allows different processes to share information. Typical real-world examples include:

Example	Description
Shell → Window Server	The shell sends a command like `"draw: Hello GUI!"`
Browser → GPU Driver	Sending render commands
System Services	Logging, event handling, networking

In Novix OS, we'll use a **message-passing IPC** system, where one process sends a buffer of bytes to another process, and the destination retrieves it through a system call.

1. The IPC Message Structure

Let's begin by defining the **IPC message** data structure used by each process.

include/process.h

```
typedef struct ipc_message {
    char data[MAX_MSG_SIZE];
    size_t size;                // size of the message
    pid_t sender;               // sender of the message
    int pending;                // 1 = true = message available
} ipc_message_t;
```

Explanation:

- Each process owns one `ipc_message_t inbox`, where incoming messages are stored.
- `data[]` holds the message bytes.
- `size` tracks the message length.
- `sender` records the sender process ID.

- `pending` indicates if a message is waiting (1) or empty (0).

This simple design allows the kernel to safely deliver messages between isolated processes.

2. The Userland IPC Interface

user/lib/ipc.c

```c
#include "../include/syscall.h"
#include "../include/types.h"
#include "../include/stdio.h"
#include "../include/log.h"

int ipc_send(pid_t pid, const void *buf, size_t size) {
    return syscall(SYS_IPC_SEND, (intptr_t)pid, (intptr_t)buf, (intptr_t)size);
}

int ipc_recv(void *buf, size_t max_size, pid_t *sender) {
    int ret = 0;
    do {
        LOG("[ipc_recv] start syscall SYS_IPC_RECV");
        ret = syscall(SYS_IPC_RECV,
                      (intptr_t)buf,
                      (intptr_t)max_size,
                      (intptr_t)sender);
        LOG("[ipc_recv] syscall SYS_IPC_RECV done! ret = %d", ret);
    } while (ret < 0);
    LOG("[ipc_recv] received msg: %s, sender = %d, len = %d",
        (const char*)buf, (intptr_t)sender, ret);
    return ret;
}
```

Explanation:

- `ipc_send()` sends a buffer to another process.
- `ipc_recv()` blocks (loops) until a message arrives.

3. Kernel System Call Integration

kernel/syscall.c

```c
case SYS_IPC_SEND:
    f->regs.a0 = sys_ipc_send((pid_t)f->regs.a0, (void *)f->regs.a1, (size_t)f->regs.a2);
    break;

case SYS_IPC_RECV:
    f->regs.a0 = sys_ipc_recv((void *)f->regs.a0, (size_t)f->regs.a1, (pid_t *)f->regs.a2);
    break;
```

These syscall cases connect userland IPC functions with the kernel's IPC system.

4. Implementing Message Sending

sys_ipc_send()

```c
int sys_ipc_send(pid_t dest_pid, const void *buf, size_t size) {
    LOG_USER_INFO("[sys_ipc_send] called: dest_pid=%d, size=%u", dest_pid, (unsigned)size);
```

```
    if (dest_pid < 0 || dest_pid >= PROCS_MAX) {
        LOG_USER_ERR("[sys_ipc_send] invalid dest pid: %d", dest_pid);
        return -1;
    }

    proc_t *dest = get_proc_by_pid(dest_pid);
    if (!dest) PANIC("[sys_ipc_send] Error with dest_pid %d", dest_pid);

    if (size > MAX_MSG_SIZE) return -2;
    if (dest->inbox.pending) return -2;

    enable_sum();
    memcpy(dest->inbox.data, buf, size);
    disable_sum();

    dest->inbox.size = size;
    dest->inbox.sender = current_proc->pid;
    dest->inbox.pending = 1;

    LOG_USER_INFO("[sys_ipc_send] message delivered: sender=%d -> dest=%d",
                  current_proc->pid, dest_pid);
    return 0;
}
```

How it works:

1. Validates destination PID and size.
2. Copies the message to the receiver's inbox.
3. Marks the inbox as pending.

5. Implementing Message Receiving

sys_ipc_recv()

```
int sys_ipc_recv(void *buf, size_t max_size, pid_t *sender_pid) {
    LOG_USER_INFO("[sys_ipc_recv] started...");

    ipc_message_t *msg = &current_proc->inbox;
    if (msg->pending == 0) return -1;

    size_t copy_size = msg->size > max_size ? max_size : msg->size;

    enable_sum();
    memcpy(buf, msg->data, copy_size);
    if (sender_pid) *sender_pid = msg->sender;
    disable_sum();

    msg->pending = 0;
    LOG_USER_INFO("[sys_ipc_recv] ended copy_size = %d", copy_size);
    return copy_size;
}
```

6. The Init Server

servers/init/init.h

```c
#pragma once

#include <stdint.h>
#include <stddef.h>

#define MAX_SERVERS 4
#define PROC_NAME_MAX_LEN 64

const char *servers[MAX_SERVERS] = {
    "window_server.elf",
    "test_shell.elf",
    NULL
};

#define NUM_SERVERS (sizeof(servers) / sizeof(servers[0]))
```

7. Sending a Message from the Shell

user/test_shell.c

```c
#include "include/stdio.h"
#include "include/string.h"
#include "include/syscall.h"
#include "include/log.h"
#include "include/ipc.h"

void main() {
    const char *msg = "draw: Hello GUI!";

    LOG("[shell] started...");
    puts("[shell] started...\n");
    LOG("[shell] sending message: %s @ %p", msg, msg);
    printf("[shell] sending message: %s @ %p\n", msg, msg);

    ipc_send(3, msg, strlen(msg));  // 3 = pid of window_server
    LOG("Shell: draw command sent.");

    for(;;);
}
```

8. Receiving Messages in the Window Server

servers/windows/window_server.c

```c
#include "include/window_server.h"
#include "../../user/include/types.h"
#include "../../user/include/stdio.h"
#include "../../user/include/string.h"
#include "../../user/include/ipc.h"
#include "../../user/include/syscall.h"
#include "../../user/include/log.h"

int main() {
```

```
    LOG("[window_server] Ready. Waiting for draw commands...");

    while (1) {
        char buffer[BUF_SIZE];
        pid_t sender;

        LOG("[window_server] Start receiving...");
        puts("[window_server] Start receiving...\n");

        int size = ipc_recv(buffer, BUF_SIZE, &sender);

        LOG("[window_server] Ready receiving...");
        printf("[window_server] Ready receiving...\n");

        if (size > 0) {
            buffer[size] = '\0';
            LOG("[window_server] Got message: %s from pid %d", buffer, sender);
            printf("[window_server] Got message: %s from pid %d\n", buffer, sender);

            if (strncmp(buffer, "draw", 4) == 0) {
                LOG("[window_server] Drawing: %s", buffer + 5);
                printf("[window_server] Drawing: %s\n", buffer + 5);
            } else {
                LOG("[window_server] Unknown command.");
                printf("[window_server] Unknown command.\n");
            }
        }
    }

    return 0;
}
```

9. Example Output

```
[shell] started...
[shell] sending message: draw: Hello GUI! @ 0x0000000001000f86
[window_server] Start receiving...
[window_server] Ready receiving...
[window_server] Got message: draw: Hello GUI! from pid 4
[window_server] Drawing: Hello GUI!
```

Summary

Component	Purpose
`ipc_message_t`	Defines message storage for each process
`ipc_send()`	Sends a message to another process
`ipc_recv()`	Waits for and receives messages
`sys_ipc_send()`	Kernel message delivery logic
`sys_ipc_recv()`	Kernel message retrieval logic
Shell » Window Server	Demonstrates IPC communication

With IPC, Novix OS can now support **client-server communication** — the foundation for window systems, background daemons, and service-oriented architecture.

`*A smile on my face - because Novix OS support client-server communication!*`