Implementation of publisher / subscriber inter process communication model at kernel level

Harcourt Butler Technical University, Kanpur

Ayush Kumar Shakya Arunima Verma
Ashutosh Singh Hemant Singh

December 9, 2022

Certificate

This is to certify that **Ayush Kumar Shakya** (Roll no. 190108017), **Arunima Verma** (Roll no. 190108014), **Ashutosh Singh** (Roll no. 190108015), **Hemant Singh** (Roll no. 190108024), students of INFORMATION TECHNOLOGY, Harcourt Butler Technical University, Kanpur are working on their project under my guidance. They have a desire for learning and I wish success in their future project work.

Project guided by
Dr. Prabhat Verma

Abstract

Inter process communication (IPC) protocols must be provided by multitasking systems in order to allow communication between processes with varying levels of privilege.

These mechanisms may be implemented in user space or at the application level, however doing so at the application level has performance concerns because it must make numerous syscalls, which add a lot of overhead. IPC at the kernel level can decrease overheads and improve security.

Additionally, current kernel-based IPC models like Message Queues require coordination between processes in order to communicate, which might have overheads [1] and prevent anonymous communication.

We believe that each of those problems can be resolved by a kernel level publisher subscriber model.

Contents

Chapter 1

Introduction

In a modern system we have hundreds of processes running at any time and they mostly communicate using application level IPC mechanisms.

The majority of the time, processes choose not to use any communication techniques and often build everything into a single programme, inflating the programmes into large monoliths that are challenging to debug.

Software can be made more reliable and scalable, perhaps across several devices, if it is created as small micro-services that utilise effective IPC mechanisms.

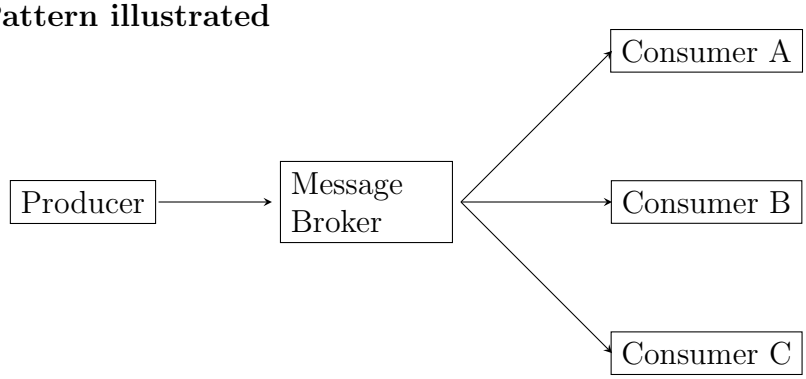
1.1 Why publisher-subscriber pattern ?

Publish/Subscribe messaging is an asynchronous service-to-service communication method used in serverless and microservices architectures. Publishers and subscribers are decoupled from one another so that publishers do not know the destination of the information that they send, and subscribers do not know the source of the information that they receive.

It has several benefits such as:

1. Decoupled/Loosely systems
2. Real-time communication
3. Ease of development

Pattern illustrated



1.2 IPC Model

1. Publishing
2. Subscription
3. Notification

1.3 Publishing

Any process may send any number of messages on a particular **topic** or multiple topics, topics may be represented by an integer.

some of the topics may be reserved for system use so programs listening to those topics can be sure that no malicious program can messages on reserved topics.

This may be implemented as a syscall at kernel level.

1.4 Subscription

Processes can subscribe to one or more than one topics and register a callback function to be called when they receive a notification. kernel may store these processes in a hash table where *key* is topic and *value* is a linked list or process identifiers or PIDs. This registration maybe implemnted as a syscall.

1.5 Notification

When a notification arrives on a particular topic, it is added to a queue, and kernel dequeues topics one at a time and sends a copy of message to each process [4].

1.6 Sharing information

1.6.1 Shared memory

Inter process communication by shared memory is a concept where two or more process can access the common memory and communication is done through this shared memory where changes made by one process can be viewed by another process.

For applications that exchange large amounts of data, shared memory is far superior to message passing techniques like message queues, which require system calls for every data exchange.

To use shared memory, we have to perform two basic steps:

1. Request a memory segment that can be shared between processes to the operating system.

2. Associate a part of that memory or the whole memory with the address space of the calling process.

1.6.2 Message Passing

Message passing can be used as a more process-oriented approach to synchronization than the "data-oriented" approaches used in providing mutual exclusion for shared resources.

Message passing model allows multiple processes to read and write data to the message queue without being connected to each other. Messages are stored on the queue until their recipient retrieves them. Message queues are quite useful for interprocess communication and are used by most operating systems.

We opted for message passing rather than shared memory because in shared memory we have to coordinate a lot between processes which has an overhead,

Furthermore modern processors are very good at copying and moving small amount of data.

1.7 Cleanup

When is process is no longer running it may be removed from the hash table so if the same process identifer is assigned to some different process it doesn't receive messages it did not subscribe to.

Chapter 2

Related Work

2.1 Apache Kafka

Apacha Kafka is distributed event broker that provides publisher-subscriber IPC mechanism at large distributed scale.It more or less achives the functionality that we want to achive.

It is built using Java and Scala and is JVM based therefore quite resourse intensive and therefore not suitable for performance critical workloads.

2.2 D-Bus

D-Bus or Desktop bus is a system bus that provides communication between processes using a single system bus it aims to provide standardization communication for linux services.

2.2.1 Operating modes

1. Request response
2. Publish subscribe

It implemented in user space and therefore cannot be used by kernel specially during booting when it needs to start multiple services and a dbus like functionality would simplify and speed up the booting process.

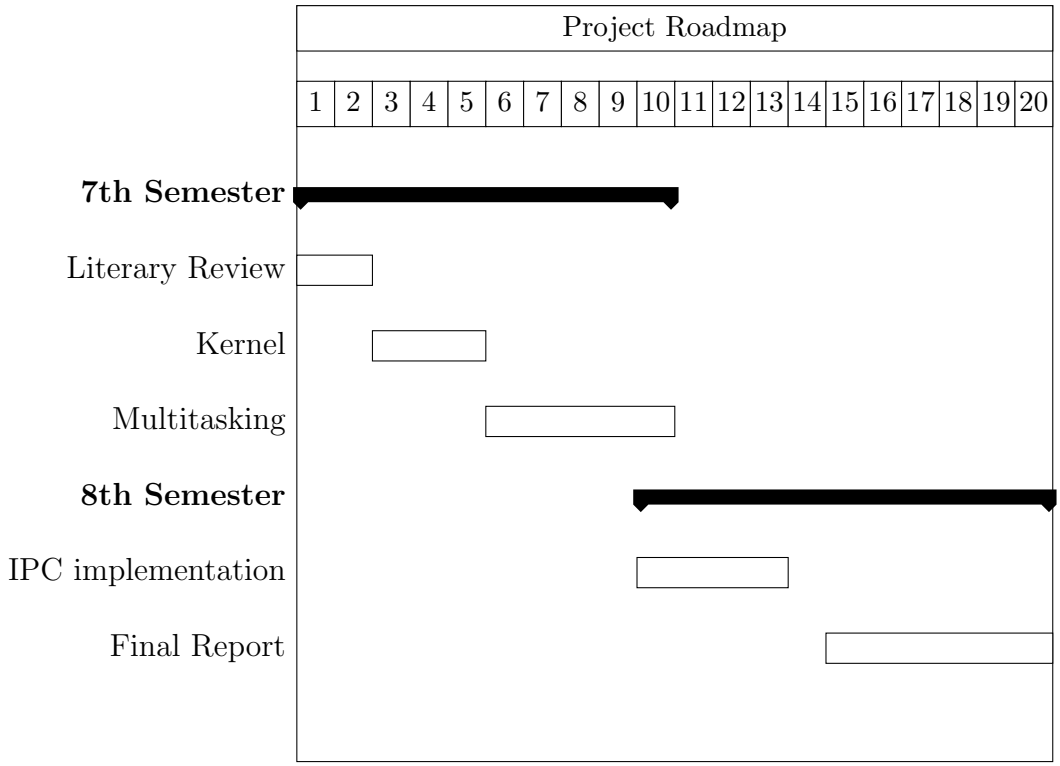
There proposals to add dbus like functionality at linux kernel level but nothing concrete is merged yet[3].

Chapter 3

Progress Report

3.1 Implementation

3.1.1 Roadmap



3.1.2 Contributions

Hemant	Literary Review and premilarly research
Arunima	Bootloader implemenation
Ashutosh	Scheduler research
Ayush	Process managment and scheduler research

3.1.3 Kernel

To reduce project scope and resource availability, we have developed a 64 bit x86 kernel written in C and will concentrate on optimising our ipc mechanisms for x86 platform only.

The kernel we developed includes a bootloader that loads the kernel and initializes virtual memory as well switches to long mode (64 bit mode).

Additionally, the kernel contains several utility functions for both the kernel and user space programmes, as well as common data structures for heap allocation.

In addition, the kernel configures the hardware timer, keyboard, and other configurations for CPU interrupts from hardware and software.

3.1.4 Bootloader

Loading Kernel

In the x86 architecture, the bootup procedure looks for a bootloader on the first sector of the booting device (a USB flash disk, hard drive, image file, etc.). This is automatically loaded and executed during the system booting process. The bootloader is responsible for loading the rest of the OS from the booting device into memory. The bootloader cannot rely upon any functionality from the OS, such as OS calls for disk I/O, or linking an object file against static libraries. However, the x86 architecture requires the presence of the Basic Input Output System (BIOS), which provides minimal terminal, keyboard and disk support. Your bootloader will use these BIOS calls to load the kernel from disk into memory, set up a stack space, and then switch control to the kernel. At this point, the OS begins running.

Memory protection

Segmentation provides a mechanism of isolating individual code, data, and stack modules so that multiple programs (or tasks) can run on the same processor without interfering with one another. Whatever memory address you

came across inside a process is actually a logical address. This logical address then converts into linear address using segmentation, then into physical address(Address on RAM) using paging. Segmentation is a mechanism for dividing the linear address space into smaller protected address spaces called segments.

Paging is a different way of accessing physical memory. Instead of memory being divided into segments it is divided into pages and frames. x86 processors normally use 3 or 4 level paging with most recent ones being able to use even 5 level paging systems. Using paging allows us to create arbitrary memory layouts and better utilize memory. It also allows us to implement more complex scenarios where each process has its own address space.

3.1.5 Process Management and Scheduler

We plan to introduce a process management and round robin scheduler to our kernel.

In order to avoid linking-related activities, we have chosen to employ dummy programmes that our kernel would interpret.

3.1.6 Program

A program for our kernel will be a static list of instructions with a fix length that will be interpreted instruction by instruction

```
typedef struct Program {
     int data[MAX_PROGRAM_LEN];
     int len;
     int ip;
} Program;
```

data will contain instructions for interpretation as well as any input for those instructions *ip* will contain an index that points to the next instruction.

3.1.7 Process management

The state of the process manager is kept behind a mutex so we don't get into a situation where our scheduler updates the state while we're executing our processes.

```
typedef struct
{
int current_pid;
uint32_t mutex_lock;
} ProcessState;

static Program processes[MAX_PROCESS_LIMIT];
static int pid_count = 0;

static ProcessState state = {
        .current_pid = -1,
        .mutex_lock = MUTEX_UNLOCKED,
};
```

The scheduler will simply run in an infinite loop run the currently active task move the instruction pointer for the current task to the next instruction and wait for a certain amount of time to make sure all processes get equal amount of execution time.

A CPU timer interrupt will periodically give control back to the kernel from the scheduler and we move to the next available task.

3.1.8 Achieving multitasking

Using techniques specified above we were able to achieve preemptive multitasking, using above techniques and a hardware timer based scheduler.

We initialized two processes with varying task lengths and tasked them to the tasked management system using **scheduler_addtask** procedure.

```
Program p1 = {
    .data = {....},
    .len =   20,
    .ip = 0
};

Program p2 = {
    .data = {....},
    .len =   4,
    .ip = 0
};

scheduler_addtask(p1);
scheduler_addtask(p2);

scheduler_start();
```

The scheduler was able to run these task concurrently and produced following output.

Output :

```
Process 1: Running
Process 1: Running
Process 2: Running
Process 2: Running
Process 1: Running
Process 1: Running
Process 1: Running
Process 2: Running
```

As predicated instructions of both processes were running concurrently and both were given equal amount of time to execute certain instructions.

Chapter 4

Work remaining

1. Actually executing instructions
2. IPC implementation
3. Real world demo
4. Final project report

Bibliography

[1] S. L. Mirtaheri, E. M. Khaneghah and M. Sharifi, "A Case for Kernel Level Implementation of Inter Process Communication Mechanisms," 2008 3rd International Conference on Information and Communication Technologies: From Theory to Applications, 2008, pp. 1-7, doi: 10.1109/ICTTA.2008.4530360.

[2] R. Rajkumar, M. Gagliardi and Lui Sha, "The real-time publisher/subscriber inter-process communication model for distributed real-time systems: design and implementation," Proceedings Real-Time Technology and Applications Symposium, 1995, pp. 66-75, doi: 10.1109/RTTAS.1995.516203.

[3] David Herrmann "Bus IPC", https://lists.linuxfoundation.org/pipermail/ksummit-discuss/2016-July/003047.html 2016

[4] Jochen Liedtke. 1993. "Improving IPC by kernel design. In Proceedings of the fourteenth ACM symposium on Operating systems principles (SOSP '93). Association for Computing Machinery", New York, NY, USA, 175–188. https://doi.org/10.1145/168619.168633