Inter Process Communication Through Shared Memory

Akshay Pawar, Manuj Khajuria and Suresh Dub

Project Report

Model Institute of Engineering and Technology
Kot Bhalwal, Jammu
181122

Inter Process Communication Through Shared Memory

Akshay Pawar, Manuj Khajuria and Suresh Dub
Department of Computer Science

Abstract

This paper deals with a study of interprocess communication in distributed systems. The interprocess communication means how the processes communicate with each other while fulfilling the users' needs. It says what the matters concerned while communicating between processes in distributed systems.

The paper describes different interprocess communication mechanisms like shared memory, message passing and remote procedure calls used in systems.This article reports the results of a research project that is exploring how to support memory sharing among multiple tasks. The scheme described here uses conventional memory management hardware in a new way. It has all the advantages of task isolation, but allows cooperating sets of tasks to share parts of their address spaces.

The chief advantage of our scheme is that it pennits tasks to share objects that contain pointers (addresses) like linked lists, even though each task usually interprets memory addresses privately. In our scheme, a task dynamically establishes an area of memory to be shared and grants other tasks access to that area. Access to objects in the shared area is no more expensive than access to objects in any other part of memory.

Introduction

In any multitasking computer system, the part of the operating system known as the task manager or process manager switches the CPU among multiple tasks. It meticulously saves the registers and machine state for one task and loads the saved machine state for another. When a multitasking system supports virtual memory, each task, sometimes called a process, executes in its own private memory address space without knowledge of the address spaces of other tasks that execute concurrently. That is, each task uses memory addresses starting at zero and extending upward. and the operating system arranges for the memory

management hardware to map each task's addresses into a different region of physical memory.

The operating system must save and restore the mappings from a task's address space to physical memory, just as it saves other machine registers, when it switches the CPU from that task to another. In a virtual memory system, each task imagines that it has all of the memory to use, starting at location zero. As we have said, memory management hardware maps the addresses a task generates into an area of physical memory reserved for that task.

We say that the task generates a virtual address to distinguish those addresses from the physical or real addresses that the underlying hardware uses. Later sections describe the mechanism most commonly used to perform the mapping. For now, it is only important to understand that each application program is written to use addresses starting at zero, and that conventional operating systems arrange to keep tasks completely isolated when they execute. The key concepts of virtual memory that we will discuss in this paper are protection and communication.

Clearly, isolating each task in a separate address space protects the task's memory, because other tasks cannot read or change data values. On some hardware, the operating system, or kernel, executes in its own virtual address space, called kernel space, which is distinct from that of any task. In such cases, the operating system must execute special instructions to address physical memory directly. The consequence of using virtual memory for protection is that tasks cannot easily move data among themselves. It is impossible, for example, to have one task assign a value to an integer variable that belongs to another task.

All communication between tasks must go through the operating system. However, placing a request with the operating system, or moving dala between the task's address space and the operating system's address space can be expensive. On most processors, a task must execute a special machine instruction known as a trap or system call to make such transfers. The system call instruction transfers control to the operating system and makes it possible for the system to access data in the task's address space. The operating system copies the data into its address space and changes the memory mapping. Later, when some other task places a system call to extract the data, the operating system must copy the data into that task's address space.

Implementation Overview

Description

We develop Algorithm of two programs here that illustrate the passing of a simple piece of memory (a string) between the processes if running simultaneously:

sharedmem1.c

--Attaches itself to the created shared memory portion and uses the

string. **Sharedmem2.c**

--simply creates the string from user input and shared memory portion.

Algorithm of sharedmem2.c (Server Side Program)

STEP – 1. In a C program first of all include all header files such as unistd.h , stdlib.h , stdio.h , string.h

Now include header file sys/shm.h to access shared memory system calls

STEP – 2 Struct shared_use_st --- creating a shared_use_st of type struct

int data_available --- variable which indicates data is available char

message[Text_SZ] --- char array to hold the input string main(){

process_running = TRUE

void *shared_memory = (void *)0

STEP – 3 shared_use_st *shared stuff --- structure which acts like a shared

memory char buffer[BUFSIZ]

int shmid --- varaiable for allocation of shared memory

shmid = shmget((key_t)1234, sizeof(struct shared_use_st) ,0666 | IPC_CREAt) --- Allocation of Shared Memory

on failure shmget () returns -1

STEP - 4 Attachment of segement using shmat() system call

Shared_memory = shmat(shmid,(void *)0, 0)

On failure shmat() returns -1
Shared_stuff = (struct shared_use_st *)shared_memory --- on successful attachment of segment

Performing write operation

While(process_running)

While(shared_stuff->data_available == TRUE)

Sleep(1)

STEP – 5 Server creates the string by asking the user an input

STEP – 6 Reads the input and stores it in buffer

STEP – 7 Writes it into the char array message[]

Check if data is available

STEP – 8 Terminate the loop by using string "end"

Process_running = FALSE --- indicates that the data is written and sets the flag

STEP – 9 Now Detach segment using shmdt() system call

Exit(EXIT_SUCCESS)

} ---end of the server program

Algorithm of sharedmem1.c (Client Side Program)

STEP – 1 In a C program first of all include all header files such as unistd.h , stdlib.h , stdio.h , string.h

Now include header file sys/shm.h to access shared memory system calls. Now include header file sys/shm.h to access shared memory system calls. **STEP – 2** Struct shared_use_st --- creating a shared_use_st of type struct int data_available --- variable which indicates data is available char message[Text_SZ] --- char array to hold the input string main(){

process_running = TRUE

void *shared_memory = (void *)0

STEP – 3 shared_use_st *shared stuff --- structure which acts like a shared memory int shmid --- varaiable for allocation of shared memory

STEP – 4 shmid = shmget((key_t)1234, sizeof(struct shared_use_st) ,0666 | IPC_CREAt) --- Allocation of Shared Memory

on failure shmget () returns -1

STEP – 5 Attachment of segement using shmat() system call

Shared_memory = shmat(shmid,(void *)0, 0)

On failure shmat() returns -1

STEP – 6 If shared_memory == (void *)-1

The exit(EXIT_FAILURE)

Shared_stuff = (struct shared_use_st *) shared_memory

Shared_stuff->data_available = FALSE

While process_running

If data available --- means data is available to read

STEP – 7 Print data

Sleep(rand() %4) Shared_stuff->data_available = FALSE --- clears the flag to show it has read the data

STEP – 8 Now Terminate the loop with the help of string “end”

STEP – 9 At the end detach the shared memory with the help of shmdt() system call or function

On failure shmdt() returns -1

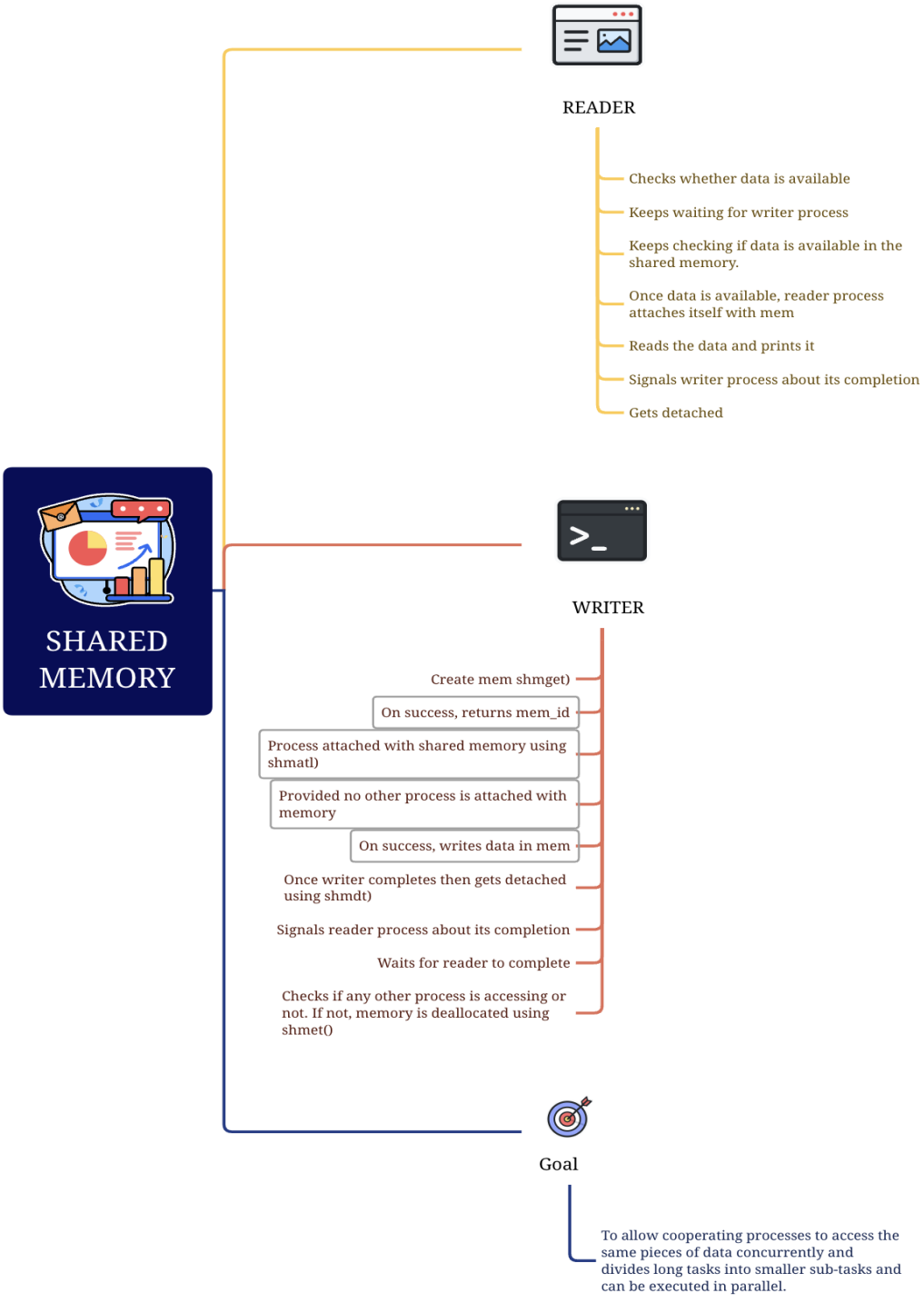
Now Deallocating the shared memory with the help of shmctl() system call

Which on failure returns -1

Exit(EXIT_SUCCESS)

} --- End of Client Process

Flowchart


SHARED MEMORY
READER
Checks whether data is available
Keeps waiting for writer process
Keeps checking if data is available in the shared memory.
Once data is available, reader process attaches itself with mem
Reads the data and prints it
Signals writer process about its completion
Gets detached
WRITER
Create mem shmget)
On success, returns mem_id
Process attached with shared memory using shmatl)
Provided no other process is attached with memory
On success, writes data in mem
Once writer completes then gets detached using shmdt)
Signals reader process about its completion
Waits for reader to complete
Checks if any other process is accessing or not. If not, memory is deallocated using shmet()
Goal
To allow cooperating processes to access the same pieces of data concurrently and divides long tasks into smaller sub-tasks and can be executed in parallel.

Conclusion

One of the hidden costs in an operating system is the time spent copying data. Communication in a multitasking, virtual memory system, whether between tasks or between a task and the operating system, usually requires copying to get data from one address space another. One way to eliminate the need for copying between address spaces is to provide a mechanism by which sections of memory can be shared among tasks and the operating system. Using conventional memory management hardware, we have proposed simple system calls that make it possible to share memory among different address spaces while still providing protection from non-cooperating tasks. Examples have shown the elegance and efficiency of the shared segment primitives.

The need of inter-process communication, some of the IPC methods, mechanisms and their implementation. We looked at IPC methods message queue, Shared memory and Semaphore made short comparisons of their properties. Each of the IPC mechanisms discussed has some advantages and some disadvantages Application Areas, features, algorithm Functionality. Each of them is a perfect resolution for a specific drawback for the applied scientist. The usability of those strategies is incredibly versatile, particularly regionally on an equivalent IPC however in numerous environments. Overall, the inter- process communication is great for introducing reliable communication, maximum performance, great functionality, application modularity and support and secure communication in our multi-process environment

References

Geeksforgeeks - https://www.geeksforgeeks.org/
Java Point - https://www.javatpoint.com/
Leetcode - https://leetcode.com/
Programiz - https://www.programiz.com/

Github

https://github.com/Suresh-Dub/OPERATING-SYSTEM-PROJECT