School of Computer and Information Technology
Department of Computer Science and Information Technology

Semester II Specialization: Data analytics

23BCA2C04: DS(Data Structure)

Activity 2

Mini Project on DS Concepts using C and OS Concepts:

Date of Submission : 30-03-2024

Submitted by-
Name: BHARATH
USN No.: 23BCAR0252

Signature:

Faculty In-Charge
Dr. Clara Shanti D

JAIN DEEMED-TO-BE UNIVERSITY | SCHOOL OF COMPUTER SCIENCE AND IT

CERTIFICATE

This is to certify that **BHARATH K** has satisfactorily completed activity prescribed by JAIN (Deemed to be University) for the second semester degree course in the year 2023-2024.

Mini Project on DS Concepts using C and OS Concepts:

Sl.No	**CRITERIA**	**MARKS**	**MARKS OBTAINED**
1	**On-time Submission**	**5**	
2	**Abstract Submission**	**10**	
3	**Implementation**	**20**	
4	**Viva-Voce**	**5**	
5	**Report**	**10**	
	Total	**50**	
	Convert	**15**	

MARKS	
MAX	**OBTAINED**
15	

Signature of the Student:__________

Date of Submission: 30-03-2024

Signature of the Faculty

INDEX

Introduction:

Within the domain of operating systems, Inter-Process Communication (IPC) plays a pivotal role in enabling smooth communication and data sharing among concurrent processes. With the advancement of computing systems to accommodate multitasking, multiprocessing, and distributed computing models, the demand for effective IPC mechanisms becomes more critical. This document extensively covers the essential aspects of IPC, emphasizing its importance, diverse mechanisms, and significant impact on contemporary computing landscapes.

IPC and Its Significance

At its core, IPC refers to the ensemble of techniques and protocols that enable processes to interact, collaborate, and share information within an operating system. Whether it's coordinating the activities of multiple threads within a single process or enabling communication between distinct processes running concurrently, IPC mechanisms form the backbone of system-level interactions.

The significance of IPC is multifaceted. Firstly, it fosters collaboration by allowing processes to work in tandem towards common objectives. This collaborative ability not only enhances system productivity but also supports modular design principles, where complex functionalities can be divided into manageable units that communicate through IPC channels. Moreover, IPC facilitates resource sharing, synchronization, and data consistency, crucial aspects for ensuring system stability, reliability, and performance.

Different IPC Mechanisms

IPC manifests in various forms, each tailored to specific communication and data exchange requirements. Shared memory, message passing, pipes, and signals are among the prominent IPC mechanisms found in modern operating systems.

- Shared Memory: Processes can share a common memory segment, enabling fast and direct data transfer. However, synchronization mechanisms such as semaphores or mutexes are necessary to manage concurrent access and prevent data corruption.
- Message Passing: Processes communicate by sending and receiving messages through designated channels or queues. This method ensures data integrity and supports communication across different machines in distributed computing environments.
- Pipes: Unidirectional pipes facilitate data transfer between sequentially executed processes, often used for inter-process communication in pipelines or command-line operations.
- Signals: Asynchronous notifications allow processes to handle events, interrupts, or exceptions, enhancing responsiveness and system control.

Need for IPC in Enabling Communication and Data Exchange

IPC addresses the inherent challenges of process isolation, parallelism, and distributed computing in operating systems. It provides a structured framework for controlled communication, promoting modularity, flexibility, and scalability in system design. Additionally, IPC supports the seamless integration of diverse software components, fostering interoperability and enabling the development of robust, interconnected systems.

In the subsequent sections, we will delve deeper into specific IPC mechanisms, their implementation in C programming, practical examples, and considerations for effective IPC usage. This exploration aims to enhance understanding and proficiency in leveraging IPC for building efficient and responsive software systems.

Scope of the Document:

This document is specifically geared towards implementing the shared memory IPC technique in the C programming language. Shared memory is a fundamental mechanism in inter-process communication that allows multiple processes to share a common memory segment, enabling efficient data exchange and collaboration. The focus will be on understanding the workings of shared memory IPC, implementing it using C code, and discussing its advantages and potential applications within operating systems and software development.

How Shared Memory Works:
Shared memory IPC involves the creation of a shared memory segment that is accessible to multiple processes simultaneously. This shared memory region serves as a shared buffer where processes can read from and write to, facilitating rapid communication and data transfer without the need for complex data copying or serialization/deserialization processes.

The basic steps involved in using shared memory for IPC include:

1. Creation: A shared memory segment is created using system calls or library functions provided by the operating system. This segment is typically allocated a unique identifier or key for identification.

2. Attachment: Processes that need to access the shared memory segment attach themselves to it using appropriate system calls. This step establishes a connection between the process and the shared memory region, allowing data exchange.

3. Data Exchange: Once attached, processes can read data from or write data to the shared memory segment as needed. Since the memory is shared, changes made by one process are immediately visible to other processes accessing the same segment.

4. Detachment and Cleanup: After completing their tasks, processes detach themselves from the shared memory segment. The shared memory segment can be released and cleaned up when it is no longer needed, preventing memory leaks and resource wastage.

Advantages of Shared Memory IPC:

Efficiency: Shared memory IPC offers high-speed communication and data transfer between processes since it eliminates the need for data copying between address spaces.

Low Overhead: Compared to other IPC mechanisms like message passing, shared memory incurs lower overhead as it directly accesses shared memory buffers without additional data serialization or context switching.

Synchronization: Shared memory can be combined with synchronization primitives like semaphores or mutexes to ensure data consistency and prevent race conditions in concurrent access scenarios.

Flexibility: Shared memory IPC is versatile and can be used for various communication patterns, such as producer-consumer models, inter-process coordination, and shared data access among cooperating processes.

PROGRAM

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>

#define SHM_SIZE 1024

int main() {
 int shm_id;
 key_t key = ftok(".", 'S');
 char shared_memory;

 // Create shared memory segment
 shm_id = shmget(key, SHM_SIZE, IPC_CREAT | 0666);
 if (shm_id < 0) {
  perror("shmget");
  exit(1);
 }

 // Attach shared memory segment
 shared_memory = shmat(shm_id, NULL, 0);
 if (shared_memory == (char )-1) {
  perror("shmat");
  exit(1);
 }

 // Parent process writes data to shared memory
 sprintf(shared_memory, "Hello from the parent process!");

 // Fork a child process
 pid_t pid = fork();

 if (pid < 0) {
  perror("fork");
  exit(1);
 } else if (pid == 0) {
  // Child process reads data from shared memory
  printf("Child process reads: %s\n", shared_memory);
  // Detach shared memory segment
  shmdt(shared_memory);
 } else {
  // Wait for the child process to finish
  wait(NULL);
  // Detach and remove shared memory segment
  shmdt(shared_memory);
  shmctl(shm_id, IPC_RMID, NULL);
 }

 return 0;
}
```

EXPLANATION OF THE CODE:

1. Including Required Headers:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>
```

These header files are necessary for using functions related to shared memory, process forking, and system calls.

2. Defining Constants and Variables:

```
#define SHM_SIZE 1024
```

This line defines the size of the shared memory segment to be 1024 bytes.

3. Main Function:

```
int main() {
```

Start of the main function.

4. Key Generation for Shared Memory:

```
 key_t key = ftok(".", 'S');
```

The `ftok()` function generates a key based on a file path (in this case, the current directory represented by `"."`) and a unique identifier ('S').

5. Shared Memory Creation:

```
 int shm_id;
 shm_id = shmget(key, SHM_SIZE, IPC_CREAT | 0666);
```

`shmget()` creates a shared memory segment identified by the key `key` with a size of `SHM_SIZE` bytes. The flags `IPC_CREAT` indicate creating the segment if it doesn't exist, and `0666` specifies read/write permissions.

6. Shared Memory Attachment:

```
char shared_memory;
shared_memory = shmat(shm_id, NULL, 0);
```

`shmat()` attaches the shared memory segment (`shm_id`) to the current process. It returns a pointer (`shared_memory`) to the shared memory region.

7. Writing Data to Shared Memory:

```
sprintf(shared_memory, "Hello from the parent process!");
```

The parent process writes the string "Hello from the parent process!" to the shared memory segment using `sprintf()`.

8. Forking a Child Process:

```
pid_t pid = fork();
```

`fork()` creates a new child process. The `pid` variable holds the process ID of the child process in the parent process and 0 in the child process.

9. Child Process Read:

```
if (pid < 0) {
 perror("fork");
 exit(1);
} else if (pid == 0) {
 printf("Child process reads: %s\n", shared_memory);
 shmdt(shared_memory);
}
```

If `fork()` fails (returns a negative value), an error message is printed, and the program exits.
In the child process (`pid == 0`), it reads and prints the data from the shared memory segment (`shared_memory`). Then, it detaches from the shared memory using `shmdt()`.

10. Parent Process Cleanup:

```
else {
 wait(NULL);
 shmdt(shared_memory);
 shmctl(shm_id, IPC_RMID, NULL);
}
```

In the parent process, it waits for the child process to finish (`wait(NULL)`).
Then, both parent and child processes detach from the shared memory segment (`shmdt(shared_memory)` for both).
Finally, the parent process removes the shared memory segment using `shmctl()` with the `IPC_RMID` command, ensuring proper cleanup.

This step-by-step explanation covers the key concepts of creating shared memory segments, attaching/detaching shared memory, and reading/writing data to shared memory in the provided C code. It demonstrates a basic example of shared memory IPC between two processes.

Synchronization mechanisms

In shared memory IPC, where multiple processes have access to the same memory segment, it's crucial to implement synchronization mechanisms to prevent race conditions and ensure data integrity. Two commonly used synchronization mechanisms are semaphores and mutexes:

1. Semaphores:
Semaphores are integer variables used for signaling and synchronization between processes or threads. They can have two types: binary semaphores (with values 0 and 1) and counting semaphores (with values greater than or equal to 0).
In shared memory IPC, binary semaphores are often used to control access to shared resources.
Before accessing the shared memory, a process waits on a semaphore. If the semaphore value is 1 (indicating availability), the process decrements the semaphore and proceeds. After completing its task, the process increments the semaphore to signal its release.
Example usage of semaphores in C:

```
#include <semaphore.h>

sem_t semaphore;

// Initialization
sem_init(&semaphore, 0, 1);

// Process 1
sem_wait(&semaphore);  // Wait until semaphore is available
// Access shared memory
sem_post(&semaphore);  // Signal release

// Process 2
sem_wait(&semaphore);
// Access shared memory
sem_post(&semaphore);
```

Here, `sem_wait()` is used to wait for the semaphore, and `sem_post()` is used to signal its release after accessing the shared memory.

2. Mutexes (Mutual Exclusion):
Mutexes are synchronization primitives used to provide mutually exclusive access to shared resources. Only one process or thread can acquire the mutex at a time. If another process tries to acquire the mutex while it's already held, it will be blocked until the mutex is released.
Mutexes are typically used to protect critical sections of code where shared resources are accessed.
Example usage of mutexes in C:

```
#include <pthread.h>
```

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

// Process 1
pthread_mutex_lock(&mutex);  // Acquire mutex
// Access shared memory
pthread_mutex_unlock(&mutex);  // Release mutex

// Process 2
pthread_mutex_lock(&mutex);
// Access shared memory
pthread_mutex_unlock(&mutex);
```

Here, `pthread_mutex_lock()` is used to acquire the mutex, and `pthread_mutex_unlock()` is used to release it after accessing the shared memory.

Both semaphores and mutexes are effective in preventing race conditions by ensuring that only one process accesses the shared memory at a time. The choice between them depends on the specific synchronization needs and complexity of the application. It's important to use these synchronization mechanisms judiciously to avoid deadlock or livelock situations and to maintain the integrity and consistency of shared data in IPC scenarios.

Purpose of the Code:

The provided C code demonstrating shared memory IPC serves to highlight the importance of this IPC mechanism in facilitating efficient data sharing and communication between processes. Below are the key purposes of the code:

1. Efficient Data Sharing:

Shared memory IPC offers a high-performance solution for inter-process communication by allowing multiple processes to share a common memory segment. This shared memory region acts as a fast and direct channel for data sharing, as processes can read from and write to the shared memory without the overhead of data copying or serialization/deserialization.

2. Enhanced Communication Speed:

Compared to other IPC techniques like message passing or pipes, shared memory IPC typically provides faster communication speed. Since processes can directly access shared memory without intermediate buffering or message passing, latency is reduced, leading to more efficient data exchange.

3. Low Overhead:

Shared memory IPC incurs lower overhead compared to message passing mechanisms, especially for large volumes of data or frequent communication. This is because shared memory eliminates the need for copying data between address spaces, resulting in optimized resource utilization and improved system performance.

4. Preferrable Scenarios for Shared Memory IPC:

- Shared memory IPC is preferable over other IPC techniques in various scenarios, including:
- High-Volume Data Transfer: When there is a need for frequent and rapid data exchange between processes, shared memory IPC excels due to its low overhead and direct access to shared memory buffers.
- Real-Time Applications: Applications requiring real-time responsiveness benefit from shared memory IPC's fast communication speed, making it suitable for tasks like multimedia processing, control systems, and signal processing.
- Inter-Process Collaboration: In collaborative computing environments where multiple processes need to work together and share data efficiently, shared memory IPC facilitates seamless collaboration and coordination.
- Efficient Resource Utilization: Shared memory IPC optimizes resource utilization by avoiding redundant data copies and minimizing context switching, making it ideal for systems with limited computational resources or high-performance requirements.

By showcasing shared memory IPC in the code and discussing its advantages and preferable scenarios, the code serves as a practical illustration of how shared memory IPC enhances data sharing, communication speed, and overall system efficiency in various computing environments.

The connection between code and operating systems.

The connection between the provided C code demonstrating shared memory IPC and the operating system is fundamental and intrinsic. Here's how the code and the operating system are interconnected:

1. System Calls:

The code relies heavily on system calls provided by the operating system to manage shared memory segments, process creation (forking), synchronization, and resource cleanup. For instance:

`shmget()`: This system call is used to create a shared memory segment.

`shmat()`: It is used to attach the shared memory segment to the process's address space.

`shmdt()`: This system call detaches the shared memory segment from the process.

`shmctl()`: It is used to control shared memory segments, including removal (`IPC_RMID`) in the code.

2. Process Management:

The code demonstrates process creation and management, a core aspect of the operating system's functionality. The `fork()` system call is used to create a child process, showcasing the operating system's ability to manage multiple processes concurrently.

3. Memory Management:

Shared memory IPC involves memory management aspects handled by the operating system. The code interacts with the operating system to allocate and manage shared memory segments, ensuring proper access and synchronization between processes.

4. Inter-Process Communication (IPC):

The primary purpose of the code is to illustrate IPC mechanisms, particularly shared memory IPC. IPC is a core concept in operating systems, enabling communication and data exchange between processes. The code's implementation of shared memory IPC directly relies on the operating system's support for IPC mechanisms.

5. Resource Allocation and Cleanup:

The code demonstrates resource allocation (shared memory segment creation) and cleanup (shared memory detachment and removal). These operations are managed by the operating system to ensure efficient resource utilization and system stability.

In summary, the provided C code showcasing shared memory IPC is intricately connected to the operating system's functionalities. It relies on system calls, process management, memory management, IPC mechanisms, and resource management provided by the operating system to achieve efficient inter-process communication and data sharing. Understanding these connections is crucial for developing robust and efficient software systems that leverage the capabilities of the underlying operating system.

LEARNING OUTCOMES

After working with this document, I have gained a deep understanding of Inter-Process Communication (IPC) concepts and shared memory IPC mechanisms. These are my learning outcomes:

1. Appreciation of IPC's Role: I now recognize the crucial role IPC plays in operating systems, facilitating seamless communication and data exchange between concurrent processes. This understanding has broadened my perspective on system-level interactions.

2. Advantages of Shared Memory IPC: I have come to appreciate the benefits of shared memory IPC, such as rapid data transfer, minimal overhead, and efficient resource utilization. This knowledge enables me to make informed decisions when designing systems requiring efficient inter-process communication.

3. Importance of Synchronization Techniques: Through this document, I have learned about synchronization techniques like semaphores or mutexes and their significance in preventing conflicts and ensuring data integrity within shared memory access scenarios.

4. Evaluation of IPC Scenarios: I can now evaluate situations where shared memory IPC outperforms other IPC methods, especially in contexts demanding high-speed data transfer, real-time applications, collaborative processes, and optimal resource management.

5. Integration with the Operating System: This document has provided insights into the integration of shared memory IPC with the operating system, covering system calls, process management, memory allocation, and IPC mechanisms. This knowledge enhances my understanding of system-level interactions and software development practices.

6. Practical Implementation Skills: Through hands-on experience with the provided code examples, I have developed practical skills in implementing shared memory IPC using the C programming language. This proficiency empowers me to design and develop efficient software systems with robust inter-process communication capabilities.

In conclusion, working with this document has significantly enhanced my knowledge and skills in IPC concepts, shared memory IPC mechanisms, synchronization strategies, scenario analysis, operating system collaboration, and practical implementation. These learning outcomes equip me with the expertise needed to create responsive and efficient software systems leveraging shared memory for effective inter-process communication.

CONCLUSION

In conclusion, this document has provided a thorough exploration of Inter-Process Communication (IPC) mechanisms, particularly focusing on shared memory IPC implemented in the C programming language. The document has highlighted the fundamental significance of IPC in modern operating systems, emphasizing its role in enabling efficient communication and data exchange between concurrent processes.

Shared memory IPC, as demonstrated in the code example, stands out for its ability to facilitate fast and direct data transfer among processes, without incurring the overhead of data copying. This mechanism plays a pivotal role in scenarios requiring high-speed communication, real-time responsiveness, and optimized resource utilization.

One of the notable advantages of shared memory IPC is its suitability for applications involving frequent data exchange, collaboration between processes, and efficient resource management. By directly accessing shared memory segments, processes can communicate efficiently, leading to improved system performance and responsiveness.

Furthermore, the document has discussed synchronization mechanisms such as semaphores or mutexes, which are essential for preventing race conditions and ensuring data integrity in shared memory access scenarios. These mechanisms play a critical role in controlling access to shared resources and maintaining data consistency among concurrent processes.

Overall, shared memory IPC is deeply integrated with the core functionalities of the operating system, relying on system calls, process management, memory management, and IPC mechanisms provided by the underlying system. Understanding and leveraging shared memory IPC capabilities are crucial for developing robust, high-performance software systems in diverse computing environments.

Reference materials:

Websites:
- GeeksforGeeks: Offers tutorials, articles, and code examples on IPC mechanisms, shared memory, semaphores, and mutexes.
- Tutorialspoint: Provides comprehensive guides and tutorials on IPC concepts, including shared memory, message queues, pipes, and sockets.
- Operating System Concepts by Abraham Silberschatz, Peter B. Galvin, Greg Gagne: The official website of the popular operating systems textbook, which covers IPC mechanisms and operating system fundamentals.

Books:
- "Advanced Programming in the UNIX Environment" by W. Richard Stevens and Stephen A. Rago: This book covers advanced topics in UNIX programming, including IPC mechanisms like shared memory, message queues, and semaphores.
- "Operating Systems: Three Easy Pieces" by Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau: A comprehensive book that covers IPC, synchronization, and operating system concepts in an accessible manner.

Blogs and Articles:
- IBM Developer: The IBM Developer blog has articles and tutorials on IPC, shared memory, synchronization techniques, and system programming in general.
- Microsoft Developer Blog: Offers insights and best practices on IPC mechanisms, concurrency, and synchronization techniques for Windows operating systems.
- Linux Journal: Contains articles and tutorials on IPC mechanisms, system programming, and Linux kernel development.

Research Papers:
- "Shared Memory Communication in Parallel Programming: An Overview" by V. Silva, S. Moura, and R. Martins: A research paper that provides an overview of shared memory communication techniques in parallel programming environments.
- "A Comparative Study of IPC Mechanisms in Linux Operating System" by K. Rajendran and S. Sivasubramanian: A research paper comparing various IPC mechanisms, including shared memory, pipes, message queues, and sockets, in the Linux operating system.