Inter-Process Communication (IPC)

In modern computer systems, multiple processes often need to communicate with each other. This communication between processes is referred to as inter-process communication (IPC). **Inter-Process Communication (IPC)** is a mechanism that allows processes to communicate with each other and synchronize their actions.

Objectives of IPC

- Enable data sharing between processes.
- Coordinate the execution of processes.
- Improve performance and efficiency through parallelism.
- Ensure data integrity and consistency.

Types of IPC Mechanisms

1. Pipes

Pipes are a unidirectional form of IPC. They enable communication between two processes, where one process writes data to the pipe, and the other reads from it. There are two main types of pipes namely:

- **Unnamed Pipes**: Used for communication between parent and child processes.
- **Named Pipes (FIFOs)**: Can be used for communication between unrelated processes.

  Example of unnamed pipe

```
#include <stdio.h>
#include <unistd.h>

int main() {
    int fd[2];
    pipe(fd);
    pid_t pid = fork();

    if (pid == 0) { // Child process
        close(fd[0]); // Close unused read end
```

```
        char msg[] = "Hello from child";
        write(fd[1], msg, sizeof(msg));
        close(fd[1]); // Close write end
    } else { // Parent process
        close(fd[1]); // Close unused write end
        char buffer[100];
        read(fd[0], buffer, sizeof(buffer));
        printf("Received message: %s\n", buffer);
        close(fd[0]); // Close read end
    }
    return 0;
}
```

2. Message Passing

In message passing, processes communicate with each other by sending messages. The messages can be sent either synchronously or asynchronously. In synchronous communication, the sending process waits for a response from the receiving process before proceeding, while in asynchronous communication, the sending process does not wait for a response.

Pros of Message Passing:

1. Security: Message passing is more secure than shared memory because messages are sent directly between processes, and only the intended recipient can access the message.
2. Flexibility: Message passing is more flexible than shared memory because it allows processes to communicate even if they are not running on the same machine or operating system.
3. Error handling: In message passing, it is easier to handle errors because each message is processed independently. If an error occurs, it is isolated to that specific message, and it does not affect other messages or the entire system.

Cons of Message Passing:

1. Overhead: Message passing has more overhead than shared memory because messages must be copied and queued for transmission.

2. Latency: In message passing, there is more latency than shared memory because messages must be queued and processed by the operating system.
3. Complexity: Message passing can be more complex than shared memory because it requires explicit coding to send and receive messages, and there is a need to handle message queues and buffering.

Message Passing in C

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct msg_buffer {
    long msg_type;
    char msg_text[100];
};

int main() {
    key_t key;
    int msgid;
    struct msg_buffer message;

    // Generate unique key
    key = ftok("progfile", 65);

    // Create message queue and return id
    msgid = msgget(key, 0666 | IPC_CREAT);
    message.msg_type = 1;

    printf("Write Data: ");
    fgets(message.msg_text, sizeof(message.msg_text), stdin);

    // Send message
    msgsnd(msgid, &message, sizeof(message), 0);

    printf("Data sent is : %s \n", message.msg_text);
```

```
    return 0;
}
```

3. Shared Memory

In shared memory, a region of memory is shared by two or more processes. The processes can read and write data to this shared memory region, and the changes made by one process are immediately visible to the other processes. Shared memory is one of the fastest IPC mechanisms because data transfer between processes is done directly in memory without any need for data copying or buffer allocation.

Pros of Shared Memory:

1. **Speed:** Shared memory is one of the fastest IPC mechanisms. Since data is shared directly in memory, there is no need for data copying or buffer allocation, making it faster than other mechanisms.
2. **Efficiency:** Shared memory is an efficient mechanism because it reduces overhead. Processes can read and write to the shared memory region directly without the need for an intermediary.
3. **Data Sharing:** Shared memory is ideal for situations where processes need to share large amounts of data. Since data is stored in a shared memory region, all processes can access it without the need for copying.

Cons of Shared Memory:

1. **Synchronization:** Shared memory requires careful synchronization between processes to avoid data inconsistency. A process must ensure that it is reading the latest data and that no other process is modifying the same data.
2. **Security:** Shared memory is not secure because it can be accessed by any process that has permission to access it. If one process has malicious intent, it can alter the data, which can cause problems for other processes that rely on that data.
3. **Management:** Managing shared memory can be challenging, especially when multiple processes are involved. The system must ensure that the shared memory region is allocated and deallocated correctly to prevent memory leaks and other problems.

Shared Memory in C

```
#include <stdio.h>
```

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

int main() {
    // ftok to generate unique key
    key_t key = ftok("shmfile", 65);

    // shmget returns an identifier in shmid
    int shmid = shmget(key, 1024, 0666 | IPC_CREAT);

    // shmat to attach to shared memory
    char *str = (char *) shmat(shmid, (void *) 0, 0);

    printf("Write Data: ");
    fgets(str, 1024, stdin);

    printf("Data written in memory: %s\n", str);

    // detach from shared memory
    shmdt(str);

    return 0;
}
```

4. Semaphores

Semaphores are a synchronization mechanism that enables multiple processes to access a shared resource. A semaphore is used to coordinate access to a shared resource, such as a shared memory region, by controlling access to the resource through a counter. It ensures that multiple processes do not access a shared resource simultaneously.

Semaphores in C

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
```

```
sem_t sem;

void* thread_func(void* arg) {
    sem_wait(&sem);
    printf("Entered...\n");

    // critical section
    sleep(4);

    printf("Exiting...\n");
    sem_post(&sem);
}

int main() {
    pthread_t t1, t2;
    sem_init(&sem, 0, 1);

    pthread_create(&t1, NULL, thread_func, NULL);
    sleep(2);
    pthread_create(&t2, NULL, thread_func, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    sem_destroy(&sem);
    return 0;
}
```

5. Sockets

Sockets are endpoints for communication between processes over a network. They include Stream sockets (TCP) and datagram sockets (UDP). Sockets use a client-server architecture, where one process acts as a server and listens for incoming connections, while the other process acts as a client and establishes a connection to the server. Below is an example of Client-Server Communication using Sockets in Python

```
# server.py
import socket
```

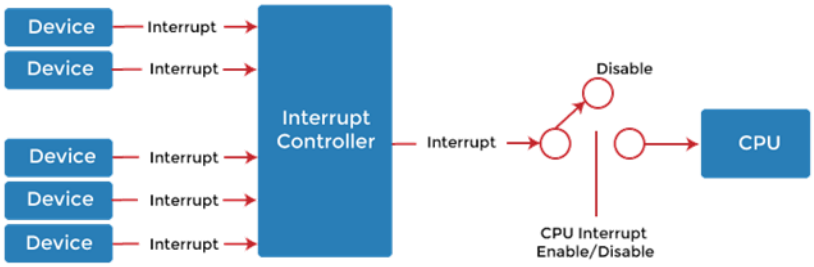
```
s = socket.socket()
s.bind(('localhost', 12345))
s.listen(5)

while True:
    c, addr = s.accept()
    print(f"Connection from {addr}")
    c.send(b"Hello from server")
    c.close()

# client.py
import socket

s = socket.socket()
s.connect(('localhost', 12345))
print(s.recv(1024))
s.close()
```

6. Signals

Signals are a mechanism for inter-process communication in Unix-based systems. A signal is an interrupt delivered to a process, which can be used to notify the process of an event or to request a specific action. Best used to handle asynchronous events like interruptions.

7. Remote Procedure Call (RPC)

RPC is a mechanism for inter-process communication that allows a process to call a function in another process, as if the function were a local function. RPC provides a higher-level abstraction than shared memory or message passing, as it enables processes to communicate using a procedure call interface.



At the CPU level, inter-process communication is typically implemented using hardware interrupts. An interrupt is a signal generated by hardware or software to interrupt the normal execution of a process and transfer control to an interrupt handler routine, which can be used to respond to the interrupt and perform some action.

In modern computer systems, interrupts are used extensively to handle a wide variety of events, such as ***I/O operations***, ***timer interrupts***, and ***system calls***. For example, when a process needs to perform an I/O operation, it sends a request to the operating system, which then issues an I/O command to the device driver. The ***device driver*** then waits for the I/O operation to complete and sends an ***interrupt*** to the operating system when it is finished. The operating system then sends a ***signal*** to the waiting process, indicating that the I/O operation has completed and the process can continue.

Interrupts are a critical component of inter-process communication at the CPU level, as they enable efficient and timely handling of events and provide a way for processes to communicate with each other and with the operating system. Interrupts are implemented in hardware, which makes them very fast and efficient, and they are an essential part of the way modern computer systems handle inter-process communication.

Each of these IPC methods has its own set of advantages and disadvantages, and the choice of method depends on the application's specific requirements. It is important to evaluate the available options carefully and choose the most appropriate method for the particular scenario.

Practice exercises

1) Pipe Communication

Create a parent and child process using pipes where the child sends a message to the parent and the parent prints the message.

2) Message Queue

Implement a message queue where one process writes messages to the queue and another process reads and prints the messages.

3) Shared Memory

Create a shared memory segment where one process writes a string and another process reads and prints it.

4) Semaphore Synchronization

Use semaphores to synchronize two threads such that they print messages alternately.

5) Socket Communication

Implement a simple client-server application using sockets where the client sends a message to the server and the server echoes it back.

Project

Objective: Compare different IPC mechanisms in terms of performance, ease of use, and suitability for different scenarios.

Requirements

1. **Implementation**: Implement a simple producer-consumer problem using pipes, message queues, shared memory, and sockets.
2. **Benchmarking**: Measure and compare the performance of each implementation.
3. **Analysis**: Analyze the results and discuss the strengths and weaknesses of each IPC mechanism.