Inter-Process Communication (IPC)

Inter-Process Communication (IPC) refers to the mechanisms that allow processes to communicate and synchronize their actions. Processes are isolated in their memory space, so to share information or cooperate, they need special communication techniques.

When a child process is created, the parent can only receive its exit status upon termination. However, processes often need to communicate while they are still running, especially if they are cooperating processes.

Cooperating Processes vs. Independent Processes

- **Independent Process:** Cannot affect or be affected by the execution of another process. They run in isolation.
- **Cooperating Process:** Can affect or be affected by the execution of another process. Cooperation allows sharing information and speeding up computations, but it also introduces potential dangers like **data corruption** and **deadlocks**.

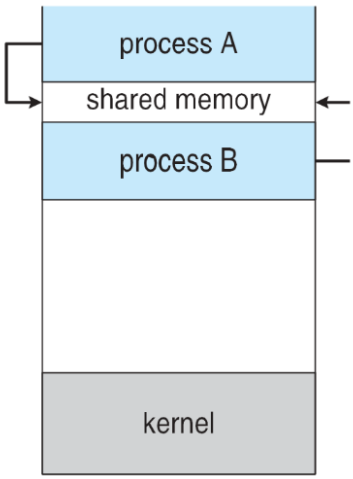
Types of Inter-Process Communication (IPC)

There are two main types of IPC mechanisms:

1. **Shared Memory**
2. **Message Passing**

Shared Memory

In shared memory IPC, an area of memory is shared between processes that wish to communicate. Unlike message passing, where the operating system manages communication, shared memory is controlled by the user processes themselves.



The key issue in shared memory is **synchronization**, as processes need to ensure they don't access the same memory at the same time in conflicting ways (e.g., reading and writing).

Producer-Consumer Problem (Cooperating Processes)

The **Producer-Consumer Problem** is a classic example of process cooperation where:

- The **Producer** process generates data.
- The **Consumer** process consumes that data.

Both processes share a buffer, where the producer places its data, and the consumer retrieves it. The challenge is to synchronize the two processes so the consumer does not consume unproduced data, and the producer does not overwrite data that hasn't been consumed.

There are two types of buffers:

1. **Unbounded buffer:** No practical limit on buffer size.
2. **Bounded buffer:** Fixed buffer size (which we will use in the example).

Bounded-Buffer Shared Memory Solution

In this solution, the producer and consumer share a memory region that includes a buffer, with the following key shared variables:

- **Buffer**: Holds the items.
- `in`: Index for where the next produced item will be placed.
- `out`: Index for where the next consumed item will be retrieved from.

```
#define BUFFER_SIZE 10

typedef struct {
    // Define item structure here
} item;

item buffer[BUFFER_SIZE];
int in = 0;  // Next position for producer
int out = 0; // Next position for consumer
```

Bounded-Buffer Producer Code

The producer will keep producing items and placing them in the buffer. If the buffer is full, the producer will wait.

```
item next_produced;

while (true) {
    /* Produce an item */
```

```
    // Wait if buffer is full
    while (((in + 1) % BUFFER_SIZE) == out);

    // Place item in the buffer
    buffer[in] = next_produced;

    // Move 'in' to the next position
    in = (in + 1) % BUFFER_SIZE;
}
```

Bounded-Buffer Consumer Code

The consumer keeps consuming items from the buffer. If the buffer is empty, the consumer will wait.

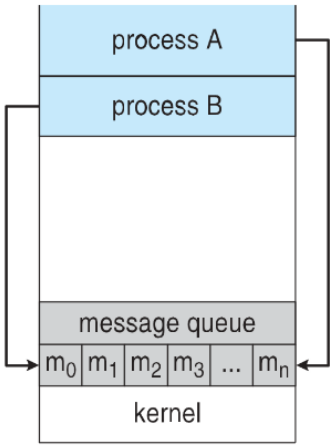
```
item next_consumed;

while (true) {
    // Wait if buffer is empty
    while (in == out);

    // Retrieve the next item from the buffer
    next_consumed = buffer[out];

    // Move 'out' to the next position
    out = (out + 1) % BUFFER_SIZE;

    /* Consume the item */
}
```

Key Points of Bounded-Buffer Solution

- **Producer waits** if the buffer is full, allowing the consumer to free up space.
- **Consumer waits** if the buffer is empty, waiting for the producer to produce new items.
- This solution only allows a maximum of `BUFFER_SIZE - 1` items in the buffer at any time.

Message Passing

Message passing is a method of exchanging messages between processes.



In operating systems, communication between processes can occur in various modes, each with distinct characteristics:

1. Simplex Communication

Simplex communication allows data to flow in only one direction. One process (the sender) can transmit data to another (the receiver), but the receiver cannot send data back.

2. Half-Duplex Communication

In half-duplex communication, data can flow in both directions, but not simultaneously. A process can send or receive data, but it must switch between these two modes.

3. Full-Duplex Communication

Full-duplex communication allows data to be sent and received simultaneously. Both processes can transmit and receive information at the same time.

4. Unidirectional Communication

Unidirectional communication refers to data flow in a single direction, similar to simplex.

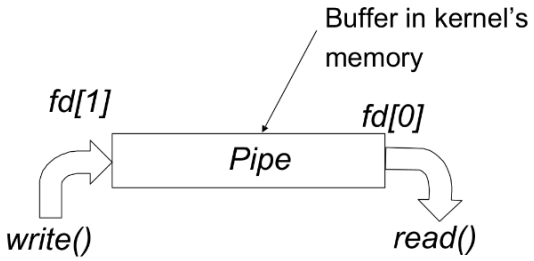
5. Bidirectional Communication

Bidirectional communication allows data to flow in both directions, incorporating both half-duplex and full-duplex methods.

Use of Pipes

Pipes allow for communication between two related processes, typically a parent and its child. Here's how it works:

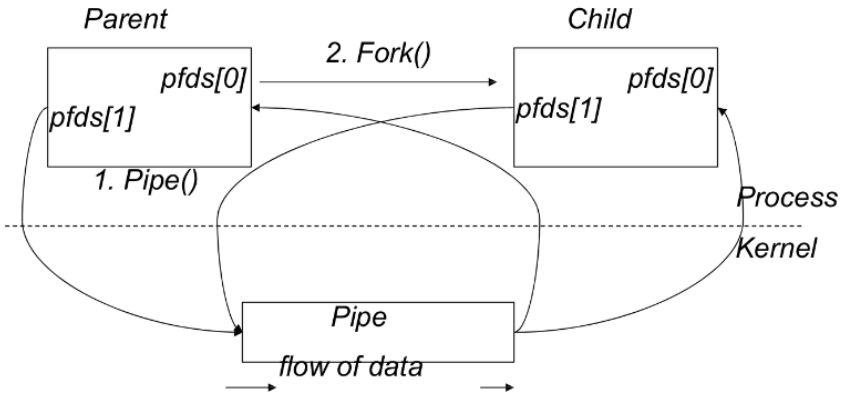
- **Creation:** A pipe is created using the `pipe()` system call, which initializes a buffer in the kernel's memory and provides two file descriptors:
  - `fd[0]` for reading from the pipe.
  - `fd[1]` for writing to the pipe.



- **Forking:** When a `fork()` is executed, the child process inherits the file descriptors. This setup allows the child to read what the parent writes to the pipe, or vice versa.

Example: Using Pipes with Environment Variables

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

int main() {
    int fds[2];

    if (pipe(fds) == -1) {
        perror("pipe");
        exit(1);
    }
    if (fork() == 0) {
        close(fds[1]);  // Child closes write end
        char buf[100];
        read(fds[0], buf, sizeof(buf));  // Read data from the pipe
        printf("Child reads: %s\\n", buf);
        exit(0);
    } else {
        close(fds[0]);  // Parent closes read end
        const char* message = "Hello from parent";
        write(fds[1], message, sizeof(message));  // Write to the pipe
    }
    return 0;
}
```



Communication Using FIFOs

FIFOs, also known as named pipes, provide a way for unrelated processes to communicate using named pipe files in the filesystem:

- **Setup:** Unlike pipes, FIFOs are created using `mkfifo()` system call, which makes a FIFO special file that can be accessed like any other file using its pathname.
- **Advantages:** Allows bidirectional communication and does not require a parent-child relationship between processes.
- Once you have created a FIFO, any process can open it for reading or writing, in the same way as an ordinary file. Opening a FIFO for reading normally blocks until some other process opens the same FIFO for writing, and vice versa.

Example:

```
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main() {
    const char* fifo_path = "/tmp/my_fifo";  // Path for the FIFO (named pip
e)

    // Create a FIFO file (named pipe) with read/write permissions for the ow
ner, group, and others
    mkfifo(fifo_path, 0666);

    // Open the FIFO for writing
    int fd = open(fifo_path, O_WRONLY);

    // Simple message to write to the FIFO
    const char* message = "Hello from the writer process!";
```

```
    // Write the message to the FIFO
    write(fd, message, sizeof(message));

    // Print the data that was written to the FIFO
    printf("Written to FIFO: %s\n", message);

    // Close the FIFO file descriptor
    close(fd);

    return 0;
}
```

```
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main() {
    const char* fifo_path = "/tmp/my_fifo";  // Path for the FIFO (named pip
e)

    // Open the FIFO for reading
    int fd = open(fifo_path, O_RDONLY);

    // Buffer to store the data read from the FIFO
    char buf[1024];

    // Read the data from the FIFO
    read(fd, buf, sizeof(buf));

    // Print the data that was read from the FIFO
    printf("Read from FIFO: %s\n", buf);

    // Close the FIFO file descriptor
    close(fd);

    return 0;
}
```

Redirection of Standard Input and Output Using `dup` and `dup2`

Standard Input (`stdin`), **Standard Output** (`stdout`), and **Standard Error** (`stderr`) are represented by file descriptors:

- `stdin` = file descriptor 0
- `stdout` = file descriptor 1
- `stderr` = file descriptor 2

Use Case: Redirecting Output to a File Using `dup` or `dup2`

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

int main() {
    // Open a file to redirect output
    int fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);

    if (fd < 0) {
        perror("open");
        return 1;
    }

    // Duplicate the file descriptor 'fd' to the lowest available descriptor,
which will be stdout (1)
    int newfd = dup(1);
    dup2(fd, 1);  // Now all output to stdout will be written to "output.txt"

    // Write to stdout (which is now redirected to the file)
    printf("This will be written to output.txt\\n");

    // Restore stdout using the saved descriptor 'newfd'
    dup2(newfd, 1);

    // Output to the console again
    printf("This will be printed to the terminal\\n");

    // Close the file
    close(fd);

    return 0;
}
```

Key Points:

- `dup`: Duplicates `fd` to the lowest available descriptor.
- `dup2`: Duplicates `fd` to a specific descriptor (`newfd`), giving you control over the redirection.

- **Redirection**: These calls can be used to redirect **stdin**, **stdout**, or **stderr** to files, pipes, or other devices, enabling flexible input/output handling between processes.