INTERPROCESS COMMUNICATION

IPC in Linux Operating Systems

Lab Manual 6 - OS Laboratory

Lab Objectives

- Understand how IPC (Interprocess Communication) happens between two processes
- Implement IPC using different methods:
  - Message Queues
  - Pipes (Named and Unnamed)
  - Shared Memory
- Execute and debug IPC programs in Linux

What is IPC?

Definition

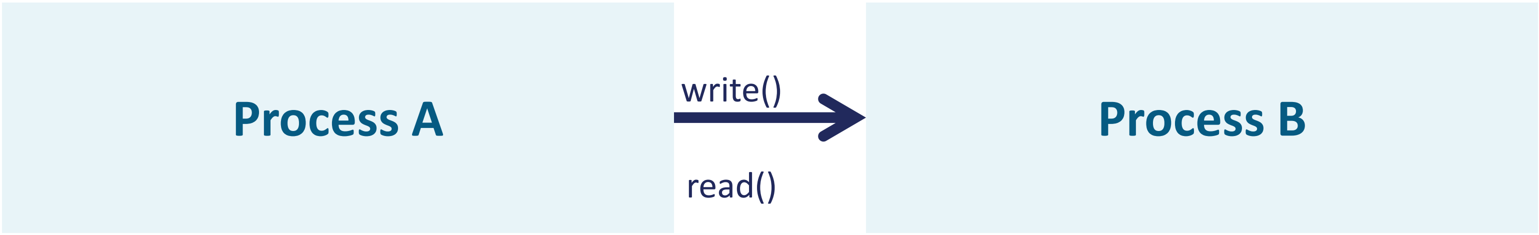
Interprocess Communication (IPC) is the mechanism whereby one process can communicate with another process and exchange data.

Three Main Methods

- Pipes
- Message Queues
- Shared Memory

Pipes - Overview

Unidirectional byte streams connecting processes



- One-way communication only
- Uses file descriptors fd[0] (read) and fd[1] (write)
- Typically used between parent and child processes

Pipe System Call

```
int pipe(int *filedes);
```

Returns

- 0 on success
- -1 on error

File Descriptors

- fd[0] - opened for reading
- fd[1] - opened for writing

Pipe - Code Example

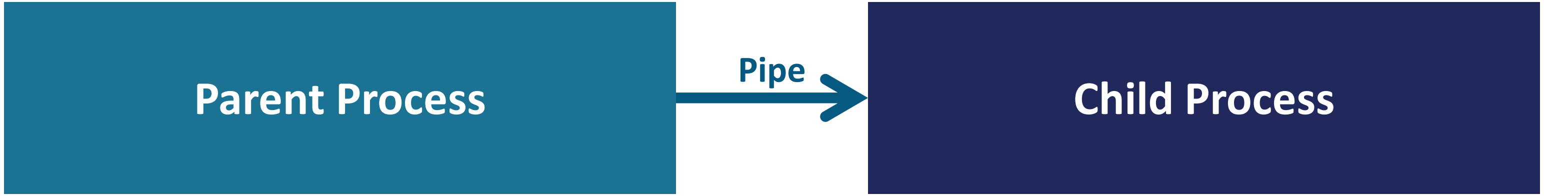
```c
int n, fd[2];
char buf[1025];
char *data = "hello... sample data";

pipe(fd);  // Create pipe
write(fd[1], data, strlen(data));  // Write to pipe

if ((n = read(fd[0], buf, 1024)) >= 0) {
    buf[n] = 0;
    printf("read %d bytes: %s", n, buf);
}
```

Pipe with Fork()

Communication between parent and child processes



- Closes read end fd[0]
- Writes to pipe via fd[1]

- Closes write end fd[1]
- Reads from pipe via fd[0]

Named Pipes (FIFOs)

Features

- Special file in filesystem
- Enables IPC between unrelated processes
- Persistent until deleted
- Created using mkfifo command or mkfifo() function

System Call

```
#include <sys/stat.h>

int mkfifo(
  const char *filename,
  mode_t mode
);
```

Message Queues

Efficient data passing between unrelated processes

Advantages over Named Pipes

- No need for synchronization of opening/closing
- Messages can be retrieved by type, not just FIFO order
- Easy and efficient information passing

Key Concept: Messages stored as linked list in kernel with unique identifiers

Message Queue Functions

msgget()

Create/access queue

msgsnd()

Send message to queue

msgrcv()

Receive message from queue

msgctl()

Control/delete queue

All functions require: #include <sys/msg.h>
Efficient data passing between unrelated processes

msgget() - Create Queue

```
int msgget(key_t key, int msgflg);
```

Parameters

- key: IPC_PRIVATE or positive integer
- msgflg: Permission flags + IPC_CREAT

```
int msgid = msgget(1234, 0666 | IPC_CREAT);
if (msgid == -1) {
    perror(\"msgget failed\");
    exit(1);
}
```

Returns

- Positive number: Queue identifier (success)
- -1: Error

msgsnd() & msgrcv()

```
// Send message
int msgsnd(int msqid, const void *msg_ptr,
           size_t msg_sz, int msgflg);

// Receive message
int msgrcv(int msqid, void *msg_ptr, size_t msg_sz,
           long int msgtype, int msgflg);
```

Message Structure Requirements

- Must start with long int (message type)
- Followed by actual data to transfer
- Must be smaller than system limit

Shared Memory

Most efficient IPC mechanism

Shared memory allows two or more processes to access the same logical memory segment. Changes made by one process are immediately visible to others.

- Most efficient data transfer method between processes
- Memory appears in each process's address space
- One process creates, others attach to it
- Changes are instantly visible to all attached processes

Shared Memory Functions

shmget()

Create shared memory segment

shmat()

Attach to address space

shmdt()

Detach from address space

shmctl()

Control/delete segment

Header: #include <sys/shm.h>

shmget() - Create Memory

```
int shmget(key_t key, size_t size, int shmflg);
```

- key: IPC_PRIVATE or integer identifier
- size: Amount of memory required in bytes
- shmflg: Permission flags (like file permissions) | IPC_CREAT
- Returns: Non-negative shared memory identifier on success, -1 on failure

```
int shmid = shmget(1234, 1024, 0666 | IPC_CREAT);
if (shmid == -1) {
    perror(\"shmget failed\");
    exit(1);
}
```

shmat() & shmdt()

```
// Attach shared memory
void *shmat(int shm_id, const void *shm_addr, int shmflg);

// Detach shared memory
int shmdt(const void *shm_addr);
```

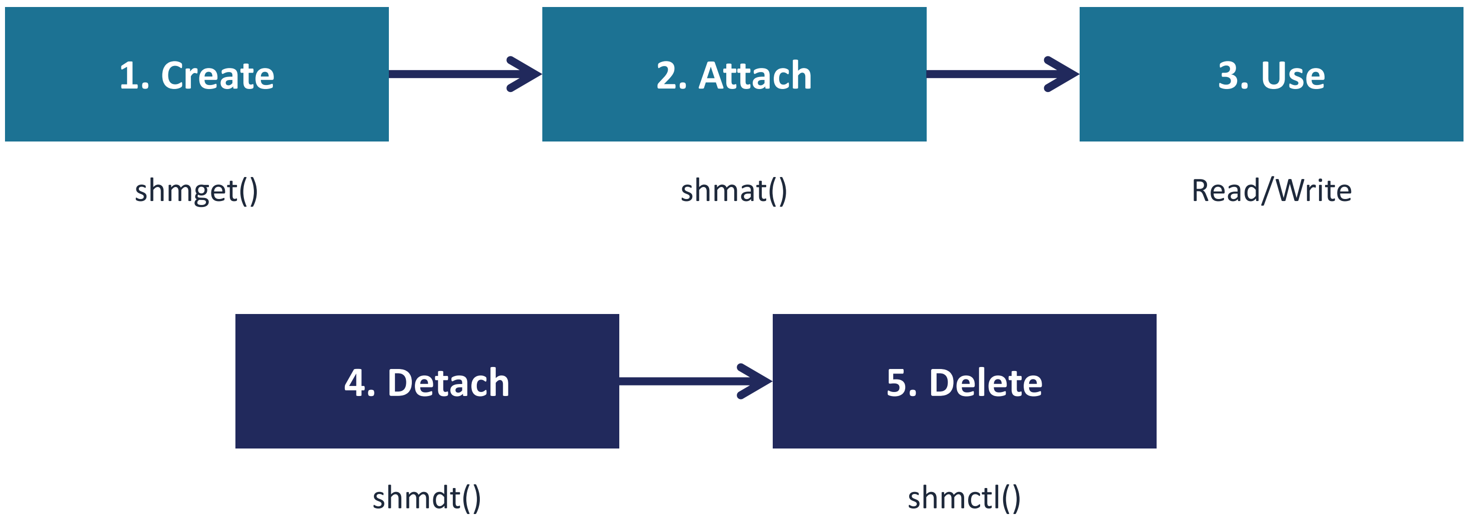
shmat() Parameters

- shm_addr: Usually NULL (let system choose)
- Returns: Pointer to memory on success, -1 on error

shmdt() Function

- Detaches memory from process
- Does NOT delete the segment
- Returns: 0 on success, -1 on error

Shared Memory Workflow



Producer-Consumer Problem

Classic synchronization problem where producer creates data and consumer uses it

Producer

- Cannot write to full buffer
- Must wait if buffer is full

Consumer

- Cannot read from empty buffer
- Must wait if buffer is empty

Readers-Writers Problem

Multiple processes sharing a resource with different access levels

Readers

- Read-only access
- Multiple readers OK

Writers

- Read and write access
- Exclusive access needed

Two Variations

- First R-W: Readers priority (no reader waits unless writer has permission)
- Second R-W: Writers priority (waiting writer blocks new readers)

Lab Exercise 1

Message Queue: Palindrome Checker

Process A sends a number to Process B via message queue. Process B checks if the number is a palindrome and responds.

- Create a message queue using msgget()
- Process A: Send integer via msgsnd()
- Process B: Receive using msgrcv(), check palindrome
- Display result and clean up with msgctl()

Lab Exercise 2

Named Pipes: Producer-Consumer

Producer writes 4 integers to FIFO queue, Consumer reads and displays them.

- Create FIFO using mkfifo()
- Producer: Open FIFO in write mode, write 4 integers
- Consumer: Open FIFO in read mode, read integers
- Display values and clean up FIFO

Lab Exercises 3 & 4

Exercise 3: Parent-Child Alphabet Exchange

Parent sends alphabet to child via shared memory. Child responds with next alphabet. Parent displays child's reply.

Exercise 4: Shared Memory Word Exchange

Producer writes words to shared memory. Consumer reads words from shared memory. Properly detach and delete shared memory after use.

Summary & Best Practices

Choose the right IPC method based on your needs:

- Pipes for simple parent-child communication
- Named Pipes for unrelated processes
- Message Queues for structured, typed messages
- Shared Memory for maximum performance

- Always clean up IPC resources (msgctl, shmctl)
- Handle errors and edge cases properly
- Consider synchronization when using shared memory

Thank You!