Shared Memory, Pipes, and Message Queues: IPC Patterns in Linux

Interprocess Communication (IPC) is essential in multitasking operating systems like Linux where processes often need to coordinate or share data. Linux offers several IPC mechanisms, each with trade-offs in speed, complexity, and use cases. Three of the most commonly used IPC methods are **Shared Memory**, **Pipes**, and **Message Queues**.

2. Shared Memory

Overview:

Allows multiple processes to access the same physical memory space. It’s the fastest IPC mechanism because it avoids kernel/user space copying once mapped.

Usage Steps:

1. Create/obtain a shared memory segment using `shmget()`.
2. Attach it to process memory space using `shmat()`.
3. Use normal memory operations to access it.
4. Detach and delete using `shmdt()` and `shmctl()`.

Example:

```
int shmid = shmget(IPC_PRIVATE, 1024, IPC_CREAT | 0666);
char *data = (char *)shmat(shmid, NULL, 0);
strcpy(data, "Shared Data");
shmdt(data);
shmctl(shmid, IPC_RMID, NULL);
```

Pros:

- Very fast data transfer.
- Efficient for large datasets.

Cons:

- No synchronization—must use semaphores/mutexes externally.
- More complex to manage lifecycle and cleanup.

Choosing the Right IPC Mechanism

Feature	**Pipes**	**Shared Memory**	**Message Queues**
Speed	Moderate	Fastest	Slower
Direction	Uni (or bi for socketpair)	Bi-directional (needs sync)	Bi-directional
Synchronization	No	External needed	Built-in
Suitable for	Simple data	Large data	Discrete messages
Related processes	Required (anonymous)	Not required	Not required

---

Conclusion

Understanding IPC patterns is crucial for building efficient and responsive systems, especially in OS-level programming, system services, and performance-critical applications. Shared memory is ideal for large, fast data exchange (with synchronization), pipes are great for simple parent-child workflows, and message queues offer flexibility and safety when message ordering and delivery are important.