.

📄 Summary: Inter-Process Communication (IPC) – Key Insights

I. What Is IPC?

Inter-Process Communication (IPC) allows processes to exchange data or signals. Essential for multitasking systems like **Linux/Unix**, it enables collaboration between:

- Parent-child processes
- Unrelated processes
- Processes across machines

II. Core IPC Mechanisms (Local Systems)

Method	Purpose / Nature	Speed	Use Case Example
1. Pipes	Unidirectional, parent-child process communication	Fast	`pipe()` with `fork()`
2. FIFOs	Named pipes for unrelated processes	Moderate	`mkfifo()`, accessed via filesystem
3. Message Queue	Queue of typed messages between processes	Scalable	`msgget()`, `msgsnd()`, `msgrcv()`
4. Shared Memory	Fastest: directly share memory	Very Fast	`shmget()`, `shmat()`
5. Semaphores	Synchronize access to shared resources	Critical	`semget()`, `semop()`
6. Signals	Event notifications (lightweight)	Instant	`signal(SIGINT, handler)`
7. Sockets	IPC over local or remote networks	Flexible	`socket()`, `bind()`, `connect()`
8. Memory-mapped files	File-backed shared memory	Fast	`mmap()`
9. D-Bus	Desktop services and application messaging	High-level	Systemd, GNOME, KDE

Best Practice:

- Use **Shared Memory + Semaphores** for fast & synchronized large data sharing.
- Use **Message Queues/Sockets** for asynchronous, flexible communications.

III. IPC in Distributed Systems

Method	Use Case / Description
TCP/UDP Sockets	Standard networking IPC (e.g., `bind()`, `listen()`)
Message Queue Servers	RabbitMQ, Kafka, ZeroMQ for scalable messaging
gRPC (RPC)	Remote function calls over network
Distributed Shared Memory	Used in HPC, e.g., OpenMPI, TreadMarks
File-based IPC (NFS)	Write/read shared files across machines
D-Bus (Extended)	Limited use over networks with TCP backend

🔁 Trade-Off:

- **Sockets/gRPC** = best for real-time, cross-machine process interaction.
- **Queue servers** = best for message durability, async, and load balancing.

IV. Low-Level Functions – How IPC Is Built in C

Common Function Groups:

Function	Role
`pipe()`	Create unnamed pipe
`fork()`	Spawn child process
`read()`	Read from pipe/shared memory
`write()`	Write to pipe/shared memory
`mkfifo()`	Create named pipe
`open()`	Open FIFO for read/write
`close()`	Close descriptor
`unlink()`	Remove named pipe/file

`popen()` & `pclose()`: High-level abstraction

- `popen("ls", "r")` opens pipe to command output.
- Used like file operations: `fgets()`, `fprintf()`.

V. Code Pattern Summary

- **Pipe + fork()**: One-way parent-child communication.
- **FIFO**: File-based IPC across unrelated processes.
- **Message Queues**: Use `msgsnd()` and `msgrcv()` for flexible message passing.
- **Shared Memory**: `shmget()` to allocate, `shmat()` to access.
- **Semaphores**: Lock critical sections with `sem_wait()` / `sem_post()`.

VI. Decision Matrix for IPC Choice

Goal	Best IPC Mechanism
High-speed data transfer	Shared Memory + Semaphores
Simple signaling	Signals
Cross-machine communication	Sockets or gRPC
Complex desktop communication	D-Bus
File-backed sharing	Memory-Mapped Files (mmap)

.

Let me know if you'd like this in a formatted DOCX or printable PDF version.