IPC: Inter-Process Communication - Comunicación entre Procesos en Sistemas Operativos

IPC (Inter-Process Communication) es el mecanismo que permite que dos o más procesos se comuniquen, sincronicen y compartan datos entre sí.

Concepto clave: Los procesos en sistemas operativos modernos están aislados en espacios de memoria separados (por seguridad). El IPC es la "puerta controlada" que permite que colaboren cuando es necesario.

¿Por qué es necesario?

Situación	Ejemplo real
Un proceso produce datos, otro los consume	Navegador descarga archivo → Explorador de archivos lo muestra
División de tareas complejas	Editor de video: un proceso renderiza, otro reproduce preview
Servicios del sistema	Aplicación solicita impresión → Servicio de impresión la gestiona
Múltiples instancias coordinadas	Navegador con varias pestañas compartiendo caché

Mecanismos de IPC

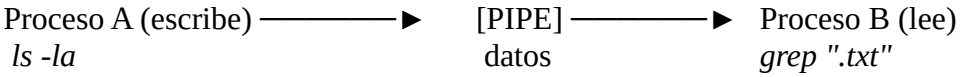
Responsabilidades del Kernel: Comunicación entre procesos (IPC)

Comunicación IPC: Pipes, señales, shared memory, sockets

Mecanismo	Descripción	Uso típico
Pipes (Tuberías)	Canal unidireccional de flujo de datos	`ls | grep txt` (comandos encadenados)
Señales (Signals)	Notificación asíncrona de eventos	Ctrl+C (SIGINT), kill (SIGTERM)
Memoria Compartida	Región de RAM accesible por múltiples procesos	Bases de datos, aplicaciones de alto rendimiento
Sockets	Comunicación entre procesos (misma máquina o red)	Servidores web, aplicaciones distribuidas
Colas de mensajes	Mensajes estructurados con prioridad	Sistemas embebidos, comunicación formal
Semáforos	Sincronización de acceso a recursos compartidos	Prevenir condiciones de carrera

Mecanismos explicados en detalle

1. Pipes (Tuberías)



Características:
- Unidireccional (uno escribe, otro lee)
- FIFO (First In, First Out)
- El kernel gestiona el buffer intermedio
- Anónimos (entre procesos padre-hijo) o con nombre (named pipes/FIFOs)

Ejemplo en shell:
bash

ps aux | grep firefox | wc -l
ps → pipe → *grep* → pipe → *wc*
Tres o mas procesos comunicándose por pipes

wc significa "word count" (contador de palabras), pero es mucho más versátil. La opción -l especifica que queremos contar líneas (lines).

2 Señales (Signals)

Notificaciones asíncronas que interrumpen un proceso para informarle de un evento.

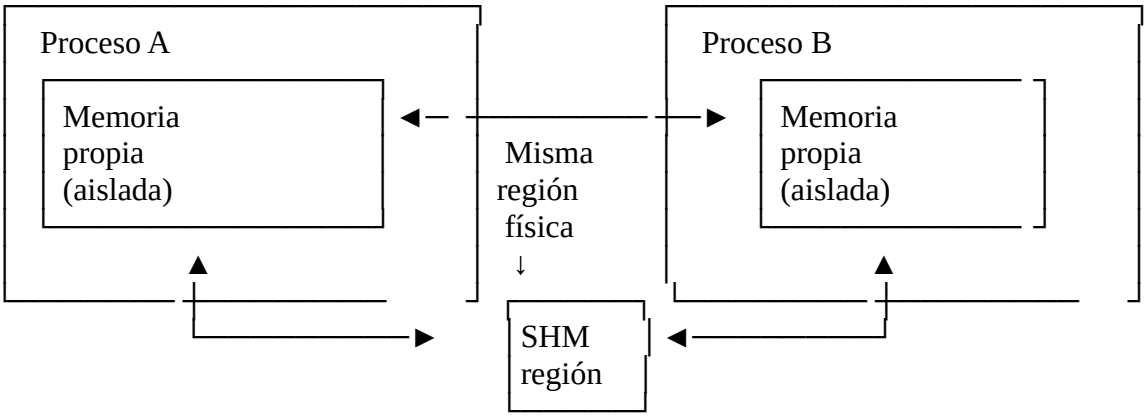
Señal	**Número**	**Significado**	**Origen típico**
SIGINT	2	Interrupción	Ctrl+C
SIGKILL	9	Terminación forzada	kill -9
SIGTERM	15	Terminación ordenada	kill
SIGSTOP	19	Pausar proceso	Ctrl+Z
SIGCONT	18	Continuar proceso	fg, bg

Enviar señales (SIGSTOP, SIGCONT) al kernel

Flujo:

Usuario presiona Ctrl+C
↓
Terminal envía SIGINT al proceso foreground
↓
Kernel interrumpe al proceso
↓
Proceso ejecuta manejador de señal (o termina por defecto)

3 Memoria Compartida (Shared Memory)



Kernel gestiona

Características:
- Más rápido que pipes/sockets (no copia datos)
- Requiere sincronización (semáforos/mutex) para evitar conflictos
- El kernel solo interviene al inicio (mapeo) y final (desmapeo)

Ejemplo:

Dos procesos shm_escritor.c y shm_lector.c independientes acceden directamente a la misma región de memoria física, sin copiar datos.

Paso	shm_escritor	shm_lector	Acción del kernel
1	shmget()	shmget()	Crea segmento de memoria compartida
2	shmat()	shmat()	Mapea memoria física al espacio virtual de cada proceso
3	Escribe "Hola..."		Sin intervención (acceso directo)
4		Lee "Hola..."	Sin intervención (misma memoria física)
5	shmdt() + shmctl(RMID)	shmdt()	Libera recursos

shm_escritor.c (Proceso A - Escribe)

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

#define SHM_SIZE 1024
#define CLAVE 1234

int main() {
   int shmid;
   char *memoria;

   // 1. Crear segmento de memoria compartida
   shmid = shmget(CLAVE, SHM_SIZE, IPC_CREAT | 0666);
   if (shmid == -1) {
      perror("shmget");
      exit(1);
   }
   printf("[Escritor] Memoria creada, ID: %d\n", shmid);

   // 2. Adjuntar (mapear) a espacio de direcciones
   memoria = shmat(shmid, NULL, 0);
   if (memoria == (char *)-1) {
      perror("shmat");
      exit(1);
   }

   // 3. Escribir directamente en la memoria compartida
   strcpy(memoria, "Hola desde escritor!");
   printf("[Escritor] Datos escritos: '%s'\n", memoria);
   printf("[Escritor] Dirección de memoria: %p\n", (void*)memoria);

   // 4. Esperar a que el lector lea
   printf("[Escritor] Esperando 5 segundos...\n");
   sleep(5);

   // 5. Desadjuntar y liberar
   shmdt(memoria);
   shmctl(shmid, IPC_RMID, NULL);  // Eliminar segmento
   printf("[Escritor] Memoria liberada. Fin.\n");

   return 0;
}
```

shm_lector.c (Proceso B - Lee)

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>

#define SHM_SIZE 1024
#define CLAVE 1234  // Misma clave que el escritor

int main() {
    int shmid;
    char *memoria;

    // 1. Obtener el segmento existente (misma clave)
    shmid = shmget(CLAVE, SHM_SIZE, 0666);
    if (shmid == -1) {
        perror("shmget (¿Ejecutaste primero el escritor?)");
        exit(1);
    }
    printf("[Lector] Memoria encontrada, ID: %d\n", shmid);

    // 2. Adjuntar a mi espacio de direcciones
    memoria = shmat(shmid, NULL, 0);
    if (memoria == (char *)-1) {
        perror("shmat");
        exit(1);
    }

    // 3. Leer directamente (misma dirección física)
    printf("[Lector] Dirección de memoria: %p\n", (void*)memoria);
    printf("[Lector] Datos leídos: '%s'\n", memoria);

    // 4. Desadjuntar (no eliminar, el escritor lo hace)
    shmdt(memoria);
    printf("[Lector] Fin.\n");

    return 0;
}
```

Compilar ambos

```
gcc shm_escritor.c -o shm_escritor

gcc shm_lector.c -o shm_lector
```

Terminal 1: Ejecutar escritor primero

./shm_escritor

Terminal 2: Ejecutar lector (mientras escritor espera)

./shm_lector

Salida esperada

Terminal 1 (Escritor):

[Escritor] Memoria creada, ID: 65536

[Escritor] Datos escritos: 'Hola desde escritor!'

[Escritor] Dirección de memoria: 0x7f8b3c7a2000

[Escritor] Esperando 5 segundos...

[Escritor] Memoria liberada. Fin.

Terminal 2 (Lector):

[Lector] Memoria encontrada, ID: 65536

[Lector] Dirección de memoria: 0x7f8a2b1c4000 ← ¡Diferente virtual!

[Lector] Datos leídos: 'Hola desde escritor!' ← ¡Mismo contenido!

Clave: Direcciones virtuales diferentes, pero misma memoria física.

Verificación con comandos del sistema

 # Ver segmentos de memoria compartida activos

ipcs -m

 # Salida típica:

 # ------ Memoria compartida Segmentos --------

 # key shmid owner perms bytes nattch status

 # 0x000004d2 65536 usuario 666 1024 2

 # nattch = 2 (dos procesos adjuntos: escritor y lector)

 # Eliminar manualmente si queda colgado:

ipcrm -m 65536

Las direcciones virtuales son diferentes (cada proceso tiene su propio espacio), pero el kernel configura la MMU para que ambas apunten a la misma memoria física real.

4 Sockets

El mecanismo más versátil: permite IPC local y en red.

Tipo	**Familia**	**Uso**
Unix Domain	AF_UNIX	Misma máquina (más eficiente)
Internet	AF_INET	Diferentes máquinas (TCP/IP)

User Space: Process → sendmsg() → Syscall → Sockets → TCP/IP → Network Device → HW

Ejemplo real: Tu navegador (proceso) se comunica con el servidor web (otro proceso, posiblemente otra máquina) vía sockets.

Ver todos los puertos TCP/UDP en escucha en Linux

netstat -tulpn

Opciones:
-t : TCP
-u : UDP
-l : solo sockets en escucha (listening)
-p : mostrar proceso/PID
-n : mostrar números, no nombres

SS equivalente más rápido a netstat

ss -tulpn

Ver solo puertos en escucha

ss -lntu

Ver conexiones establecidas (comprobar con apache, las ips de servidor y cliente)

ss -tn state established

Ver procesos asociados (necesita sudo)

sudo ss -tulpn

Estado	Significado	Ejemplo
LISTEN	Esperando conexiones entrantes	Servidor web
ESTAB	Conexión establecida	Cliente conectado
SYN-SENT	Intentando conectar	Cliente iniciando
SYN-RCVD	Recibió solicitud de conexión	Servidor respondiendo
TIME-WAIT	Esperando para cerrar	Conexión recién terminada
CLOSE-WAIT	Esperando que la aplicación cierre	
UNCONN	Creado, sin bind ni connect	Socket recién creado

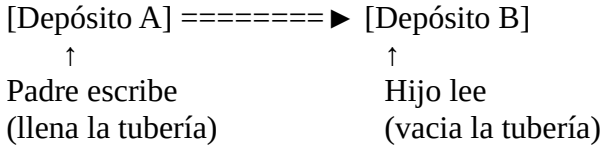
Comparativa de mecanismos IPC

Característica	Pipe	Señal	Mem. Compartida	Socket
Velocidad	Media	Alta (solo aviso)	Muy alta	Media
Dirección	Unidireccional	Asíncrona	Bidireccional	Bidireccional
Complejidad	Simple	Simple	Compleja	Media
Datos	Flujo de bytes	Solo número (señal)	Estructura libre	Flujo o datagramas
Red	No	No	No	Sí

Ejemplo práctico: Productor-Consumidor

Demostración de IPC (Inter-Process Communication) usando pipes, uno de los mecanismos que el kernel proporciona para que procesos se comuniquen.

PIPE - TUBERÍA DE AGUA

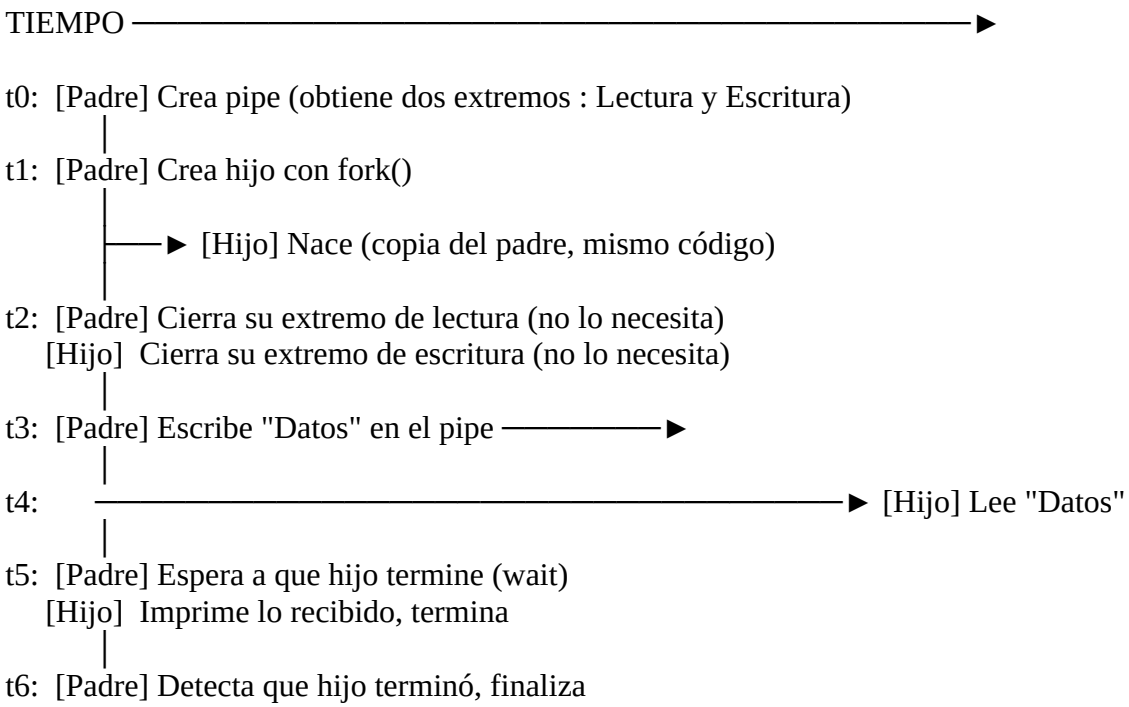


- Un solo sentido (unidireccional)
- Si está vacía, quien lee espera
- Si está llena, quien escribe espera

¿Quiénes participan?

Proceso	Rol	Acción
Padre	Productor	Crea el pipe, crea al hijo, envía datos
Hijo	Consumidor	Recibe datos del pipe, los procesa

Secuencia esperada



Clave: Son procesos independientes con espacios de memoria separados. No pueden compartir variables directamente.

Ver video

https://www.youtube.com/watch?v=Y2mDwW2pMv4

pipe_ejemplo.c

Un proceso padre envía datos a un proceso hijo a través de un pipe (tubería), que es un canal de comunicación unidireccional gestionado por el kernel.

```
#include <stdio.h>      // printf
#include <stdlib.h>     // exit
#include <unistd.h>     // pipe, fork, write, read, close
#include <sys/types.h>  // pid_t
#include <sys/wait.h>   // wait

int main() {
 int pipefd[2];
```

```
pid_t pid;
char buffer[10];

// SYSCALL: Crear pipe
if (pipe(pipefd) == -1) {
perror("Error al crear pipe");
exit(1);
}

// SYSCALL: Crear proceso (fork)
pid = fork();

if (pid < 0) {
// Error
perror("Error en fork");
      exit(1);
   }

   if (pid > 0) {
      // ========== PROCESO PADRE (Productor) ==========
      printf("[Padre] PID: %d, Hijo PID: %d\n", getpid(), pid);

      close(pipefd[0]);  // Cerrar extremo de lectura (no lo usa)

      // SYSCALL: Escribir en pipe
      write(pipefd[1], "Datos", 5);
      printf("[Padre] Enviado: 'Datos'\n");

      close(pipefd[1]);  // Cerrar extremo de escritura

      // SYSCALL: Esperar a que el hijo termine
      wait(NULL);
      printf("[Padre] Hijo terminado. Fin.\n");

   } else {
      // ========== PROCESO HIJO (Consumidor) ==========
      printf("[Hijo] PID: %d, Padre PID: %d\n", getpid(), getppid());

      close(pipefd[1]);  // Cerrar extremo de escritura (no lo usa)

      // SYSCALL: Leer de pipe (bloquea si está vacío)
      read(pipefd[0], buffer, 5);
      buffer[5] = '\0';  // Terminar string para printf

      printf("[Hijo] Recibido: '%s'\n", buffer);

      close(pipefd[0]);  // Cerrar extremo de lectura

      exit(0);  // SYSCALL: Terminar proceso hijo
```

```
    }

    return 0;
}
```

Compilamos con gcc:

gcc pipe_ejemplo.c -o pipe_ejemplo

Si hay errores, instalar gcc primero:

sudo apt update
sudo apt install gcc build-essential

Ahora lo ejecutamos:

./pipe_ejemplo

Para ver las syscalls reales que hace la aplicación que acabamos de crear:

Instalar strace si no lo tienes

sudo apt install strace

Ver syscalls en tiempo real

strace ./pipe_ejemplo

O solo contar cuáles usa

strace -c ./pipe_ejemplo

¿Qué system calls se usan aquí?

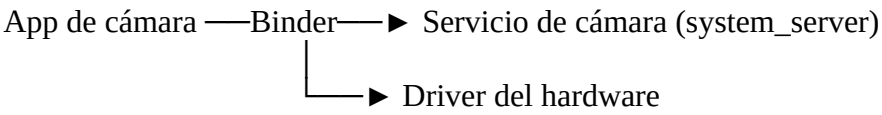
- pipe() → Crear canal IPC
- fork() → Crear proceso
- write() / read() → Transferir datos
- close() → Liberar recursos

IPC en sistemas modernos

Android: Binder IPC

Android reemplazó los mecanismos tradicionales de Linux por Binder:

- Más eficiente para móviles (batería, memoria limitada)
- Seguridad basada en permisos (cada app es usuario Linux diferente)
- Usado para toda comunicación con servicios del sistema



Windows: Mecanismos propietarios

- COM/DCOM: Objetos distribuidos
- Named Pipes: Similar a Unix
- Mailslots: Comunicación unidireccional broadcast
- Memory-mapped files: Equivalente a shared memory

Problemas clásicos en IPC

Problema	Descripción	Solución
Condición de carrera	Dos procesos modifican datos simultáneamente	Semáforos, mutex
Deadlock	Procesos se bloquean mutuamente esperando	Orden de adquisición de recursos
Starvation	Un proceso nunca accede al recurso	Prioridades, envejecimiento
Buffer overflow	Productor más rápido que consumidor	Colas circulares, control de flujo

Resumen

Mecanismos IPC

Simples	Avanzados
Pipes	Memoria compartida
Señales	Sockets
Colas mensajes	RPC/RMI
	Binder (Android)

Gestionados por el Kernel

- Crear canales
- Sincronizar acceso
- Verificar permisos
- Copiar datos entre espacios de memoria

Preguntas de comprensión

1. ¿Por qué el PDF menciona IPC como responsabilidad del kernel y no del software de sistema?
   - Pista: ¿Quién controla la memoria y los recursos compartidos?
2. Si los pipes son unidireccionales, ¿cómo logra una conversación bidireccional entre procesos?
   - Respuesta: Usando DOS pipes ($A \rightarrow B$ y $B \rightarrow A$).
3. ¿Qué mecanismo de IPC usarías para que un servidor web (nginx) se comunique con una base de datos (PostgreSQL) en la misma máquina?
   Opciones: Unix domain sockets (más rápido) o TCP sockets (más flexible).
4. ¿por qué las operaciones IPC requieren "buffering en kernel space"?
   Respuesta: Porque los procesos no comparten memoria directamente; el kernel media.