Laboratorio Práctico: Mecanismos IPC

WSL (Ubuntu)

Basado en el documento: IPC - Inter-Process Communication

OBJETIVO

Observar y analizar los tres mecanismos de IPC del documento: **Pipes**, **Señales** y **Memoria Compartida**.

PARTE 1: Pipes - Tuberías

Ejercicio 1.1: Pipe en Shell (sin programar)

bash

```
# Ejecutar comando encadenado del documento
ps aux | grep bash | wc -l
```

Pregunta: ¿Cuántos procesos participan aquí?

Que instrucción usamos para verificar que son procesos diferentes?

Ejercicio 1.2: Compilar y ejecutar pipe_ejemplo.c

Vamos a trabajar con el codigo que esta en el documento pipe_ejemplo.c

```
#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
```

Observar y responder:

Pregunta	Tu observación
¿Cuántos mensajes imprime el programa en total?	
¿El mensaje del padre siempre aparece antes que el del hijo? Ejecuta 3 veces y verifica.	
¿Qué pasa si el padre no hace wait(NULL)? (Predice, si quieres modificas)	

PARTE 2: Señales - Signals

Ejercicio 2.1: Señales desde teclado

bash

```
# Abrir una terminal y ejecutar:
sleep 300

# Presionar Ctrl+Z

# Reanudar en foreground:
fg

# Ahora presionar Ctrl+C
```

Cual(es) de las instrucciones son señales?

Explicar que hace cada linea y como afecta al proceso principal.

Ejercicio 2.2: Señales con kill

bash

```
# Terminal 1: Iniciar proceso en segundo plano
sleep 300 &
# Anotar el PID que muestra, ej: [1] 1234
```

 # Terminal 2: Enviar señales al PID
Ejemplo :
kill -2 1234

Tabla a completar:

Señal	Número	¿Qué hace?	¿El proceso puede ignorarla?
SIGINT	2		
SIGSTOP	19		
SIGCONT	18		
SIGTERM	15		
SIGKILL	9		

PARTE 3: Memoria Compartida - Shared Memory

Teoría del documento

"Región de RAM accesible por múltiples procesos" - Más rápido que pipes/sockets (no copia datos).

Ejercicio 3.1: Compilar programas del documento

bash

 # Compilar ambos programas del documento
gcc shm_escritor.c -o shm_escritor
gcc shm_lector.c -o shm_lector

Ejercicio 3.2: Ejecutar en orden correcto

bash

 # Terminal 1 - Ejecutar PRIMERO
./shm_escritor

 # Terminal 2 - Ejecutar SEGUNDO (mientras escritor espera)
./shm_lector

Que pasa si se ejecuta primero *shm_lector* y despues *shm_escritor*? Explica lo que crees que pasa sin ejecutarlo y después compruebas el comportamiento ejecutandolo.

Ejercicio 3.3: Verificar con comandos del sistema

bash

```
# En una tercera terminal, mientras ambos corren:
ipcs -m

# Salida esperada (similar a página 6):
# key        shmid      owner      perms      bytes      nattch     status
# 0x000004d2 65536      usuario    666        1024       2

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

Ejercicio 3.4: Análisis de direcciones

Pregunta	Respuesta del documento
¿Las direcciones son iguales?	
¿El contenido leído es el mismo?	
¿Por qué son diferentes las direcciones pero el contenido es igual? Ó ¿Por qué son diferentes las direcciones pero el contenido es diferente?	

PARTE 4: Comparativa de Mecanismos

Preguntas de aplicación:

1. **Para comunicar dos procesos en la misma máquina que necesitan máxima velocidad:**
   - ¿Qué eliges según el documento?
2. **Para comandos encadenados en shell (ls | grep):**
   - ¿Qué usa el sistema?
3. **Para un navegador web hablando con un servidor en otra ciudad:**
   - ¿Qué necesitas?

ENTREGABLE DEL LABORATORIO

Capturas requeridas:

1. Ejecución de pipe_ejemplo mostrando mensajes de padre e hijo
2. Salida de ipcs -m con nattch = 2
3. Ejecución de shm_escritor y shm_lector mostrando direcciones diferentes pero contenido igual

Respuestas escritas:

- Cual mecanismo crees que es más rápido, explica el porque?