72.11 Sistemas Operativos

Trabajo Práctico N°1- Inter Process Communication

Grupo N°8

Profesores:

- Godio, Ariel
- Gleiser Flores, Fernando
- Aquili, Alejo Ezequiel
- Mogni, Guido Matías

Integrantes del grupo:

- 62272 Nicolás Valentín Arias
- 63162 Santiago Mesa Rubio

ÍNDICE

Decisiones tomadas durante el desarrollo

Durante el desarrollo del sistema de cómputo distribuido, se tomaron varias decisiones clave en cuanto a los mecanismos de comunicación entre procesos (IPC, Inter-Process Communication) y las técnicas de sincronización utilizadas. A continuación se detalla cada una de ellas:

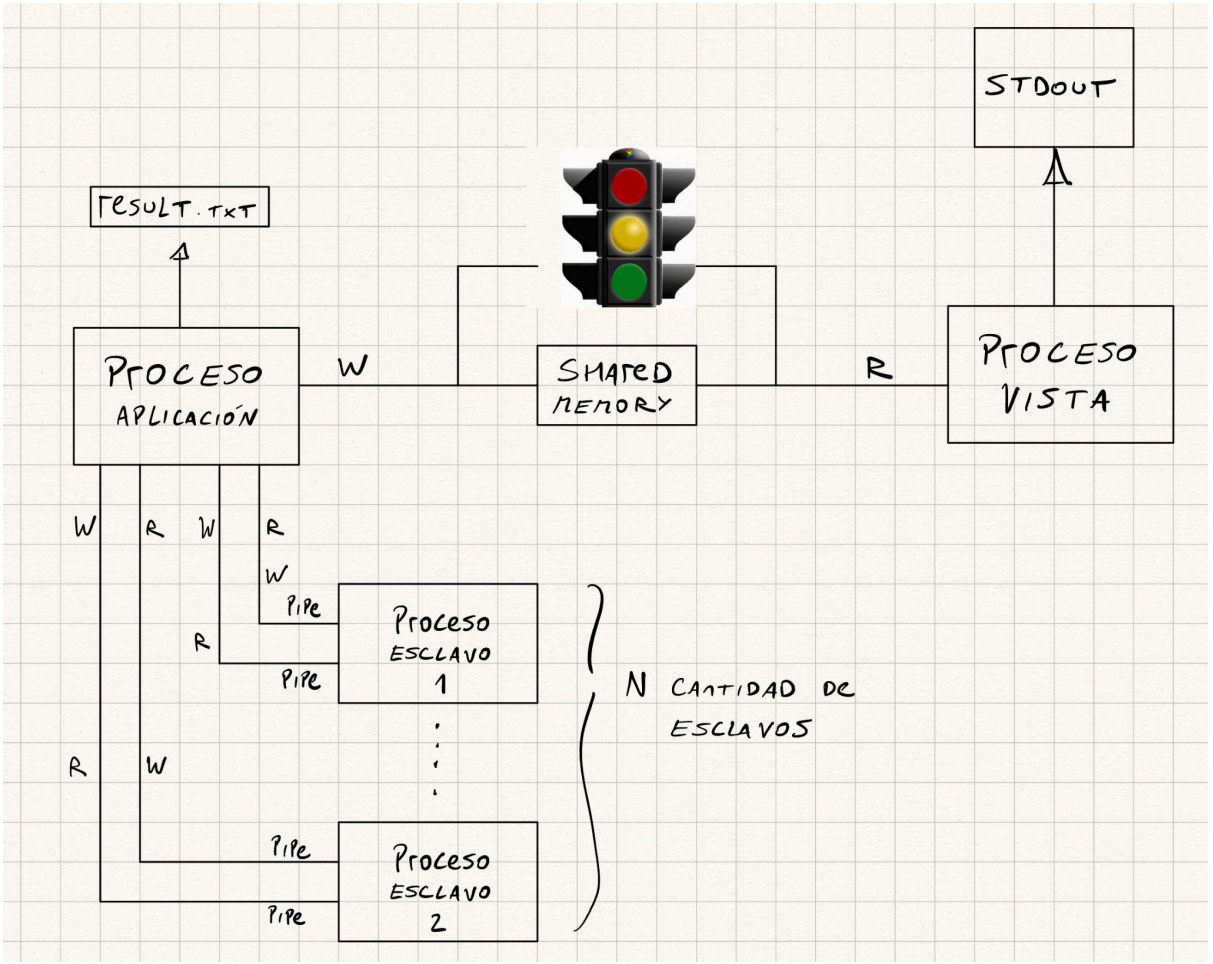
- **Pipes anónimos:** Se decidió utilizar *pipes anónimos* (`pipe(2)`) para la comunicación entre el proceso principal (aplicación) y los procesos esclavos. Los pipes se utilizaron en pares: un pipe para enviar archivos desde la aplicación a los esclavos y otro para recibir los resultados procesados de los esclavos hacia la aplicación. Esta decisión se tomó por la simplicidad y eficiencia que ofrecen los pipes en escenarios de comunicación unidireccional y de corto plazo. El uso de `select(2)` permitió la lectura simultánea de múltiples pipes, permitiendo a la aplicación gestionar eficientemente la distribución y recolección de tareas.

- **Memoria compartida (Shared Memory):** Para la interacción entre el proceso de la *vista* y la aplicación, se optó por utilizar memoria compartida (`shm_open`) en combinación con semáforos. La memoria compartida permite que ambos procesos accedan a un área de memoria común donde la aplicación va almacenando los resultados procesados (archivo, MD5 y PID del esclavo), mientras que la vista los lee y los presenta al usuario. Esta técnica minimiza el overhead de copia de datos entre procesos y facilita la comunicación entre ellos de forma rápida.

- **Semáforos:** Para controlar el acceso concurrente a la memoria compartida, se implementaron *semáforos* (`sem_open`). Esto fue necesario para evitar condiciones de carrera al momento de acceder al buffer compartido. Cada vez que la aplicación actualiza el buffer, el semáforo se libera para permitir que el proceso vista acceda a los datos y los imprima. La elección de semáforos fue crucial para garantizar la sincronización sin la necesidad de implementar técnicas de "busy waiting", mejorando la eficiencia del sistema.

- **fork() y exec():** Los procesos esclavos fueron generados mediante la llamada `fork()`, creando copias del proceso padre que posteriormente ejecutan el programa esclavo a través de `exec()`. Esta decisión permitió que la aplicación distribuyera las tareas de forma concurrente entre varios procesos hijos que trabajan de manera independiente en el procesamiento de los archivos.

Diagrama Ilustrativo de conexión de procesos



Instrucciones de compilación y ejecución

El proyecto fue desarrollado en C y requiere de un entorno compatible con POSIX. A continuación se presentan las instrucciones para compilar y ejecutar el sistema:

- Compilación:

Un *Makefile* fue utilizado para facilitar la compilación del sistema, asegurando que todos los binarios se generen correctamente sin la necesidad de invocar manualmente los comandos de `gcc`. El comando `make` compilará los siguientes binarios:

- Aplicación (`app`)
- Esclavo (`slave`)
- Vista (`view`)

Para compilar los binarios, ejecutar el siguiente comando en la terminal:

```bash
make
```

- Ejecución:

El sistema se ejecuta en tres partes: la aplicación principal, el proceso de vista (opcional) y los procesos esclavos que se generan dinámicamente. A continuación se detallan las instrucciones para ejecutar el sistema:

- Ejecutar la aplicación: La aplicación se encarga de gestionar los archivos y coordinar los esclavos.

```bash
./app file1.txt file2.txt file3.txt ...
```

- Ejecutar la vista (opcional): La vista permite visualizar los resultados en tiempo real a medida que los esclavos procesan los archivos. Puede ejecutarse de forma paralela o utilizando un pipe:

- En paralelo:

```bash
./app file1.txt file2.txt & ./view <info_de_conexion>
```

- Con pipe:

```bash
./app file1.txt | ./view
```

Limitaciones

A pesar de que el sistema está diseñado para ser robusto y eficiente, existen algunas limitaciones que fueron detectadas durante el desarrollo:

- **Número de procesos esclavos:** El número de procesos esclavos se determina dividiendo la cantidad de archivos entre diez, lo que puede ser subóptimo para grandes volúmenes de archivos. Si se procesan miles de archivos, el sistema podría

requerir demasiados procesos, lo que podría saturar los recursos del sistema operativo.

- **Tamaño del buffer en memoria compartida:** El tamaño del buffer de memoria compartida está limitado a 4096 bytes. En escenarios donde se procesen grandes cantidades de archivos o archivos de gran tamaño, esta limitación podría requerir ajustes, o la implementación de un buffer circular, lo que no se consideró en la presente versión.

- **Dependencia de la ejecución de la vista:** Si bien la vista es opcional, la aplicación debe esperar un cierto tiempo (2 segundos) para que la vista se conecte, lo que podría generar retrasos si la vista no es iniciada rápidamente.

Problemas encontrados durante el desarrollo y cómo se solucionaron

Durante el desarrollo del proyecto, se presentaron diversos problemas técnicos que fueron resueltos mediante ajustes en el código y la implementación de mecanismos adecuados:

- **Lectura de múltiples pipes simultáneamente**: Inicialmente, la aplicación intentaba leer de todos los pipes en orden secuencial, lo cual generaba problemas de sincronización. La solución fue implementar el uso de `select(2)` para monitorear múltiples pipes simultáneamente, permitiendo a la aplicación reaccionar de inmediato cuando un esclavo finalizaba el procesamiento de un archivo.

- **Sincronización entre procesos con memoria compartida:** Al principio, se observaban condiciones de carrera cuando tanto la aplicación como la vista intentaban acceder al buffer compartido simultáneamente. Para resolver esto, se implementaron semáforos (`sem_t`) que garantizaron que solo un proceso accediera al buffer en un momento dado.

- **Manejo de memoria dinámica:** Un error frecuente durante el desarrollo era el uso incorrecto de `getline()` para leer los archivos desde `stdin`, lo que provocaba fugas de memoria al no liberar los buffers correctamente. Este problema se resolvió utilizando `free()` al final de la ejecución de cada proceso esclavo.

- **Gestión de pipes al crear los esclavos:** Inicialmente, al crear los procesos esclavos con `fork()`, algunos pipes no se cerraban adecuadamente, lo que causaba que los esclavos heredaran descriptores de archivos innecesarios. Esto se solucionó cerrando explícitamente los extremos no utilizados de los pipes después de hacer la duplicación (`dup2`).

Conclusión

El sistema desarrollado cumple con los objetivos del trabajo práctico, utilizando diversos mecanismos de IPC disponibles en POSIX, como pipes anónimos, memoria compartida y semáforos, para distribuir eficientemente el cálculo de los hash MD5 entre múltiples procesos. Se lograron implementar las funcionalidades de sincronización entre procesos sin incurrir en deadlocks o condiciones de carrera. A pesar de las limitaciones mencionadas, el sistema funciona correctamente dentro del contexto planteado por el trabajo práctico.