Trabajo Práctico Nº1 - Inter Process Communication

Instituto Tecnológico de Buenos Aires - Sistemas Operativos (72.11)

Grupo 19

Ignacio Searles	**Augusto Barthelemy Solá**	**Santiago Bassi**
isearles@itba.edu.ar	abarthelemysola@itba.edu.ar	sabassi@itba.edu.ar
64.536	64.502	64.643

16 de septiembre de 2024

Resumen

El siguiente informe expone los apectos fundamentales del Trabajo Práctico Nº1 - Inter Process Communication (IPC). El trabajo consiste en la distribución y manejo de trabajos, por un proceso master, a diferentes procesos slaves; como así también, la subsiguiente visualización de las salidas de los trabajos mediante un proceso de vista.

1 Decisiones de diseño

1.1 Master

Se optó por construir la aplicación en funciones pequeñas, lo que permite una mayor versatilidad y modularidad en el diseño. Inicialmente, a cada esclavo se le asigna una cantidad de archivos configurada mediante la macro NUM_FILES_IN_INITIAL_JOB. A medida que los esclavos completan su trabajo, se les asignan nuevos archivos uno por uno hasta que todos han sido procesados. Este enfoque se eligió para distribuir trabajo correctamente en situaciones en las que un esclavo recibe archivos pesados y otro archivos livianos, lo que podría causar que uno termine rápidamente mientras el otro continúa trabajando, desperdiciando así capacidad de procesamiento.

Además, los resultados generados por los esclavos se almacenan tanto en un archivo de salida como en un buffer compartido mediante el uso de memoria compartida. Esto permite que, si el proceso view se ejecuta, pueda acceder a la información necesaria para mostrar los datos en tiempo real.

Finalmente, el código incorpora un manejo básico de errores en cada etapa crítica, como la creación de pipes, la duplicación de descriptores de archivos y la creación de procesos. Esto garantiza que el sistema pueda reaccionar adecuadamente ante posibles fallos, minimizando su impacto y facilitando la depuración.

1.2 Slaves

Los procesos esclavos son los encargados de calcular el hash MD5 de los archivos. Para ello, emplean el programa md5sum, que toma como parámetro el nombre de un archivo y genera su hash correspondiente. Hemos optado por que los esclavos lean los nombres de los archivos a procesar desde la entrada estándar y, posteriormente, impriman el hash resultante en la salida

estándar. Este enfoque permite que, una vez que el esclavo está en ejecución, podamos enviarle nuevos archivos a medida que finaliza con los anteriores, lo cual no sería posible si los archivos se proporcionaran como parámetros al iniciar el proceso.

1.3 View

El proceso view se encarga de leer los datos que el proceso aplicación escribe en el buffer de memoria compartida y mostrarlos en pantalla. En el buffer de memoria compartida guardamos estructuras con la información de salida de cada archivo procesado. Al utilizar memoria compartida, para evitar problemas con condiciones de carrera, se utilizaron semáforos como mecanismo de sincronización.

Además, se decidió leer los nombres del semáforo y del buffer desde la entrada estándar, lo que confiere mayor flexibilidad al proceso view, permitiendo su reutilización en futuros escenarios con cambios mínimos. view también puede recibir el nombre del semaforo/memoria compartida como argumento.

1.4 Comunicación entre procesos

El proceso master se comunica con los procesos slaves mediante pipes. Por cada slave existen dos pipes, un pipe de entrada al slave y otro pipe de salida del slave. El master utiliza select(2) para determinar si alguno de los pipes de salidas tiene datos para ser leídos, podiendo así determinar cuando el slave termina de procesar un archivo del trabajo. Luego, el proceso master se comunica con el proceso view mediante un shared buffer, para evitar condiciones de carrera se emplean semáforos. (Ver figura 1)

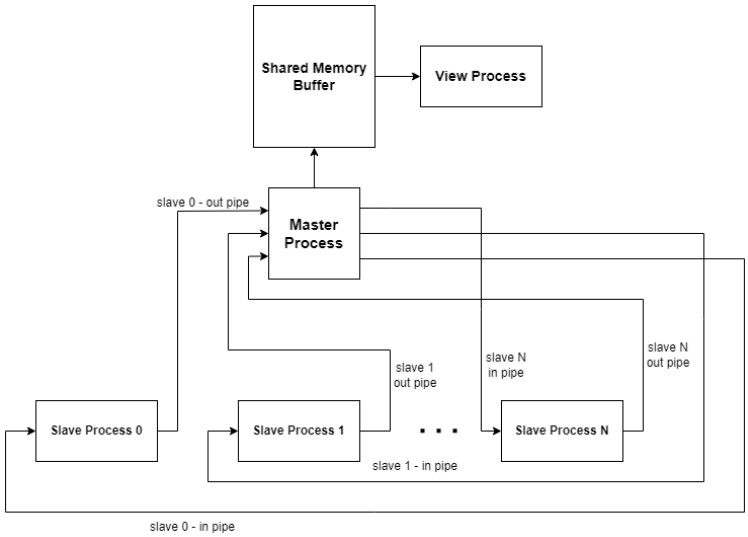

Figura 1: Diagrama donde se observa la comunicación entre procesos.

Cuando hay una nueva estructura cargada al buffer, desde el proceso app se hace un sem_post, view se queda esperando en un sem_wait hasta que haya más datos a leer. Para la condición de cierre de view, se agrega una estructura con campo pid en 0, esto supone un último sem_post desde app para la lectura de esta estructura desde view. Está condición de cierre es necesaria, pues sem_wait se queda bloqueado aunque el semáforo se cierre.

Cabe destacar que usamos el mismo nombre para tanto el semáforo como para el buffer compartido, esto no resulta un problema porque, si bien la memoria compartida y el semáforo se almacenan en el mismo lugar, para los semáforos se agrega un prefijo "sem." automaticamente.

2 Compilación y ejecución

El trabajo práctico fue compilado y ejecutado utilizando la imagen de docker provista por la cátedra. Dentro del contenedor puede ejecutarse los siguientes comandos para compilar el proyecto.

```
$> make clean
$> make
```

Una vez compilado, puede ejecutarse el programa principal mediante el siguiente comando.

```
$> ./app file1 file2 ... fileN
```

Para también correr el proceso de vista, junto con el programa principal, usando un pipe se puede ejecutar el siguiente comando.

```
$> ./app file1 file2 ... fileN | ./view
```

El proceso vista puede ser ejecutado de forma independiente del proceso principal, si el proceso principal está corriendo, se puede ejecutar el proceso vista y ingresar el nombre del semaforo por entrada estandar (provisto por salida estandar cuando se ejecuta ./app). El proceso principal espera, por defecto, 2 segundos para que se conecte el proceso vista.

```
$> ./view
/md5view
```

El proceso vista también puede recibir el nombre del semáforo/memoria compartida como argumento.

```
$> ./view /md5view
```

Siendo /md5view el nombre asignado al semaforo y al objeto de memoria compartida (modificable en app.h)

3 Limitaciones

Una de las principales limitaciones del sistema es la eficiencia en la asignación de trabajos. Actualmente, el sistema asigna archivos a los esclavos de uno en uno después de una carga inicial, lo cual puede no ser óptimo cuando los archivos tienen tamaños similares. Para mejorar el rendimiento y la utilización de los recursos, especialmente en entornos con archivos de tamaños uniformes, sería beneficioso implementar un mecanismo más dinámico para la asignación de trabajos.

Otra limitación relevante es la flexibilidad en el manejo de errores. Aunque el sistema incluye un manejo básico de errores, fortalecer la capacidad para detectar y gestionar fallos en la comunicación entre el proceso principal y los esclavos podría aumentar la robustez del sistema. Incorporar mecanismos de reintento o procedimientos de recuperación ayudaría a minimizar el impacto de errores y a mejorar la estabilidad general del sistema.

Además, el diseño actual del sistema está limitado a recibir únicamente una lista de archivos. Una mejora significativa sería permitir que el sistema acepte directorios completos, procesando todos los archivos contenidos en cada uno de los subdirectorios. Esta expansión en la funcionalidad permitiría una mayor flexibilidad y eficiencia en el manejo de grandes conjuntos de datos organizados en estructuras de directorios.

4 Problemas enfrentados

Uno de los problemas enfrentas fue el de manejar la sincronización en app con los esclavos. Como planteamos un diseño donde cada trabajo puede tener más de una tarea (en este caso procesar un archivo), tuvimos inicialmente el problema de que los esclavos procesaban una cantidad variable de tareas cada vez que checkeabamos el pipe desde main. Por este motivo planteamos un seguimiento del trabajo actual para cada esclavo. Teniendo en cuenta la cantidad de tareas a realizar y la cantidad de tareas realizadas.

Otro problema enfrentado fue determinar desde view cuando app terminaba de procesar todos los archivos. El problema es que sem_wait es una llamada bloqueante, por más que cerremos el semáforo sem_wait quedaba bloqueado. Para solucionar este problema, introducimos una estructura nula que indica que app terminó.