Sistemas Operativos

Informe TP 1

Grupo 9

Lucas David Perri - 62746

Santiago Rivas Betancourt - 61007

Franco Panighini - 61258

Índice

Decisiones

Durante el desarrollo de la aplicación, se emplearon dos métodos de comunicación entre procesos: pipes y memoria compartida. En lugar de optar por named pipes se prefirieron los pipes anónimos debido a su mayor conveniencia y a la posibilidad de evitar confusiones en el código generado por la asignación de nombres. Los pipes utilizados son de un solo sentido.

Se definió una cantidad fija de trabajadores para la aplicación mediante un define WORKER_MAX, que se fijó en 6 en función de lo discutido en el foro (debates). A cada trabajador se le envía una carga inicial de dos archivos a procesar. Si la cantidad de archivos a procesar es menor que el doble de la cantidad máxima de trabajadores establecida, se creará un trabajador por cada par de tareas. Para facilitar la lectura y escritura atómica de los datos, se asignaron dos pipes por cada trabajador.

Para la funcionalidad de los semáforos y la memoria compartida, se crearon librerías específicas para cada uno con el objetivo de evitar la repetición de código. Se eligió utilizar el modelo POSIX en lugar del modelo SYSTEM-V para la memoria compartida, así como para los mecanismos de sincronización, dado que el modelo POSIX es más sencillo de utilizar que los unnamed, que se basan en memoria. Para la sincronización de los semáforos entre Vista y Aplicación, se optó por un método de lectura desfasada, donde solo uno hace el post y el otro hace el wait, evitando así el bloqueo de Aplicación. Además, en el caso particular en el cual el proceso Aplicación termina antes que el Vista, se implementó un semáforo que hace que el proceso Aplicación espere a que el proceso Vista termine. De este modo, el proceso Aplicación no cierra prematuramente la memoria compartida con el proceso Vista.

Diagrama

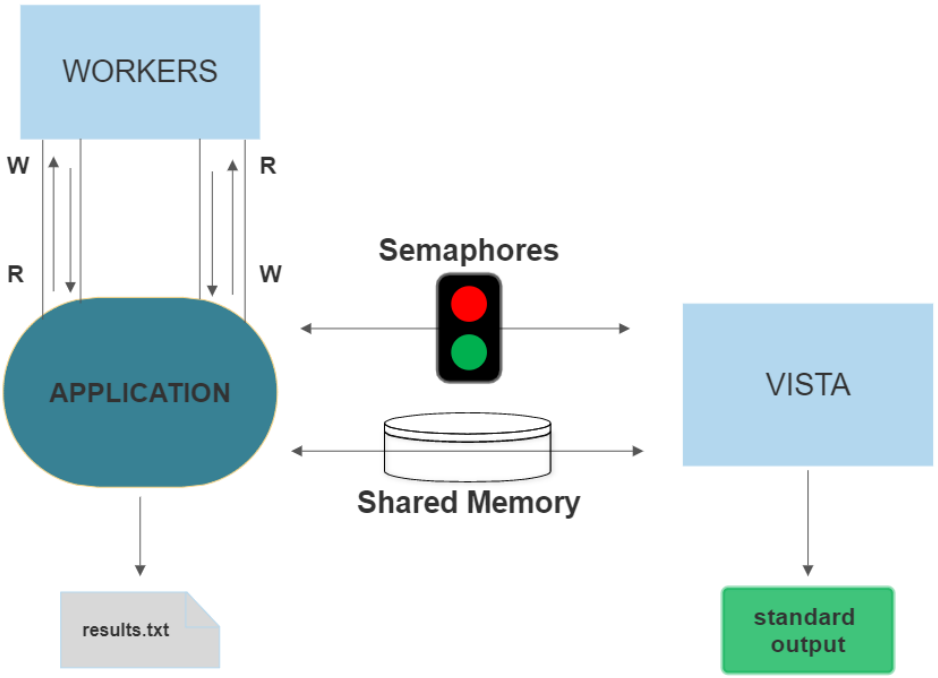

Diagrama ilustrando cómo se conectan los diferentes procesos

Instrucciones de compilación y ejecución

Se encuentran en el readme de la branch “main” en el repositorio de github del proyecto.

Limitaciones

Una limitación con la cual nos encontramos fue la cantidad de trabajadores máxima. Como los CPUs tienen un cantidad finita de cores y de poder de procesamiento, tener más trabajadores no necesariamente implicaría una mejora de performance del programa. Por esto se colocó un límite superior a la cantidad de trabajadores.

Problemas y Soluciones

El problema principal con el que nos encontramos fue en el uso de getline. Si bien logramos hacer que el programa funcione inicialmente utilizando la función getline en el main.c de aplicación, nos causaba un problema de buffering. Creemos que el problema ocurría porque la función getline hace un read “largo” en el cual se agrupaban varios hashes juntos. Además creemos que lo leído

por getline se guarda en un buffer interno, por lo que el select no detecta inmediatamente lo leído y no se desbloquea. En principio la solución encontrada fue cambiar el getline por dos reads consecutivos, uno que leía la cantidad de caracteres a leer por el segundo read. Esta solución no nos terminó de gustar, por lo que decidimos consultar con la cátedra sobre la solución propuesta. La devolución por parte de la cátedra nos resaltó la ineficiencia y "desprolijidad" de utilizar dos reads consecutivos, por lo que decidimos intentar solucionar el problema inicial con la función getline. Terminamos dándonos cuenta que si hacíamos que los archivos que se leen del trabajador sean unbuffered el problema se solucionaba.

Otro obstáculo encontrado fue la correcta apertura y cerrado de los pipes para cada uno de los trabajadores. A medida que se abría cada pipe para los trabajadores, se debió tener el cuidado de cerrarlos antes de crear un nuevo trabajador (hacer el fork) para evitar cualquier tipo de error por tener estos pipes abiertos. Para resolver este problema se utiliza la función workers_spawn. Esta función se encarga de crear todos los trabajadores y asignarles su debido pipe a cada uno. Para lograrlo se abren todos los pipes que se van a necesitar y luego en cada trabajador, después de hacer los debidos dups, se cierran todos los pipes. El abrir todos los pipes en el padre es necesario porque se necesitan guardar los pipes de manera que se pueda leer más adelante la información que viaja por ellos. La función se creó de manera modular para que se puedan crear un número variable de trabajadores. Para verificar que esta solución sea efectiva se realizó un chequeo utilizando lsof y se verificó que cada trabajador tenía un par de pipes abiertos, utilizados para la comunicación con el proceso aplicación.

Código

Todo el código del trabajo práctico es de autoría propia o perteneciente a librerías del estándar gnu-11 de C.

Bibliografía

- Manual de Linux (man)