SO - 2do cuatrimestre 2022

Trabajo Práctico

Inter Process Communication (IPC)

Grupo 2

Gastón Alasia, galasia@itba.edu.ar, 61413
Matías Della Torre, mdella@itba.edu.ar, 61016
Patricio Escudeiro, pescudeiro@itba.edu.ar, 61156

12 de septiembre de 2022

Introducción

El objetivo del Trabajo Práctico N1 consiste en la implementación de un sistema que distribuye el cómputo del hash md5 de múltiples archivos entre varios procesos. Para ello se utilizan los distintos tipos de mecanismos presentes en un sistema POSIX.

Decisiones tomadas durante el desarrollo

Para la realización del presente Trabajo Práctico, se hizo uso de dos mecanismos de Inter Process Communication (IPC): pipes anónimos y shared memory. Estos están presentes en los sistemas POSIX. Para la sincronización de procesos, se utilizaron semáforos y select.

Pipes anónimos

Para la comunicación entre el proceso aplicación (master) y los procesos esclavos (slaves), se decidió la utilización de pipes anónimos como bien indicaba el enunciado, además de que fueron los utilizados tanto en las clases teóricas como prácticas por lo que se tenía abundante información,

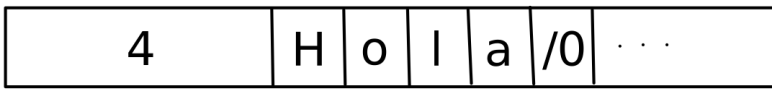
El proceso master se comunica con cada uno de sus hijos (slaves) mediante dos pipes anónimos: uno de escritura (sendPipe), en el que el padre le envía al hijo el nombre del archivo a procesar, y otro de lectura (receivePipe) en el que el hijo le devuelve al padre el hash junto con otra información relevante del mismo. Cabe destacar que para cada slave entonces existen dos pipes comunicándolo con el master.

Shared memory

La shared memory fue el mecanismo elegido para comunicar al proceso aplicación (master) con el proceso vista (view) ya que era lo requerido por el enunciado.

El proceso aplicación, al iniciarse, crea una zona de memoria compartida y le indica su nombre al proceso vista por salida estándar. Luego, a medida que escribe la información en el archivo resultado, también lo hace en la memoria compartida para que el proceso vista pueda ser consumidor de dicha información.

Cabe destacar que para hacer más eficiente el proceso de lectura desde vista, se decidió que el proceso aplicación escriba en el buffer compartido un entero que indica la longitud de un string y luego dicho string. De este modo, la vista sabe exactamente cuántos caracteres debe leer. Un ejemplo de esto es el siguiente:



De manera análoga, en la primera posición de la memoria compartida el proceso aplicación escribe la cantidad de archivos que recibió por argumento.

Semáforos

Los semáforos sirven como mecanismo de sincronización entre los procesos aplicación y vista. Se decidió que un solo semáforo binario era suficiente a la hora de indicarle al proceso vista cuándo puede leer de la memoria compartida y cuándo no (esto es, cuando el proceso aplicación está escribiendo).

Select

La función select fue utilizada como mecanismo de sincronización entre el master y los slaves, ya que sino era imposible la creación de un consumidor que no se bloqueara a la espera de recibir información de un pipe determinado.

Uso de TADs

Para mejor modularización del código, se decidió implementar dos Abstract Data Types (ADTs o TADs) que engloban las principales funcionalidades de IPC: masterADT y shmADT.

El primero corresponde a las funcionalidades del proceso aplicación: contiene métodos para crear un master y administrar tanto la creación de los esclavos mediante fork como la administración su comunicación con el master.

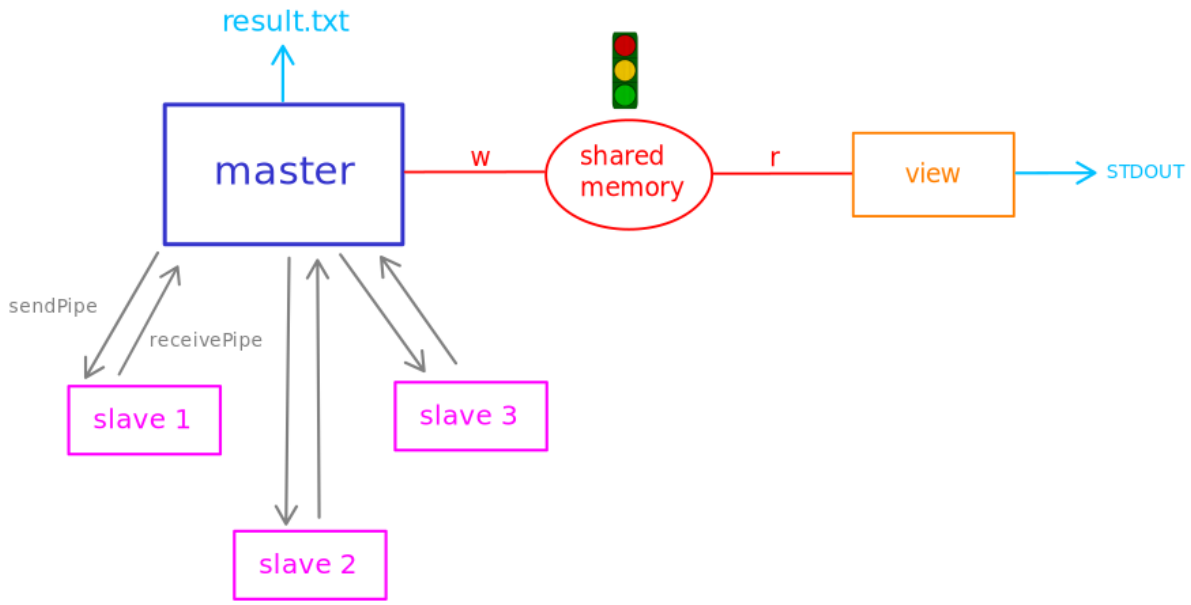
El segundo corresponde a las funcionalidades de shared memory: ofrece métodos para crearla, mapearla, obtener el puntero a su dirección virtual, escribir y leer de ella.

Cabe destacar que ambos TADs también ofrecen métodos para liberar sus recursos.

Cantidad de esclavos

Para un manejo eficiente del tiempo de ejecución del programa, se decidió utilizar el número provisto por el ejemplo de la cátedra, es decir un máximo de 5 esclavos. En caso de recibir un número grande de archivos como input, se le irán asignando a cada uno de estos esclavos a medida que terminen de procesar un archivo previo. Cabe destacar que en caso de haber menos de 5 archivos, se crearán únicamente tantos esclavos como archivos.

Diagrama de conexión de procesos



Instrucciones de compilación y ejecución

Requerimientos

Tanto para la compilación como para la linkedición es necesario utilizar Docker con la imagen de la cátedra de Sistemas Operativos del ITBA.

Para esto primero instalar Docker. Luego bajar la imagen con el comando *docker pull agodio/itba-so:1.0* y finalmente iniciar el contenedor con *docker run -v "${PWD}:/root" --privileged -ti agodio/itba-so:1.0*.

Compilación

La compilación y linkedición del siguiente proyecto se realiza mediante Makefile.

Basta con ejecutar el comando *make* en la carpeta raíz del proyecto para que éste compile y linkedite automáticamente. Se crearán tres binarios: *solve* (aplicación), *slave* (esclavo) y *view* (vista).

Ejecución

Para correr el programa, ejecutar cualquiera de los siguientes comandos en la carpeta raíz del proyecto:

- */.solve file1 file2 … file_n | ./view*

- Primero /.solve file1 file2 … file_n que imprimirá shm_name por salida estándar, y luego, en otra terminal o en foreground, /.solve shm_name.

Limpieza de binarios

Para limpiar tanto los binarios como los archivos objeto .o, ejecutar `make clean`.

Limitaciones

Entre las principales limitaciones, se encuentran la longitud de los nombres de archivo que pueden recibirse. Si bien se le asignó un buffer lo suficientemente grande, no se puede garantizar que no haya un filename de mayor tamaño. Además, el tamaño máximo del nombre de la shared memory es de 50 caracteres.

Además, de haber algún problema con cualquiera de las syscalls, el programa aborta indicando por STDERR el mensaje de error. No intenta recuperarse del error de otra manera.

Por útlimo, el nombre de la shared memory es un string constante hardcodeado en el binario. No parece correcto consultarle al usuario por terminal un nombre para un recurso compartido propio del programa.

Problemas encontrados durante el desarrollo

- En un primer momento, fue difícil interpretar concretamente qué pedía el enunciado del Trabajo Práctico. Luego de varias lecturas por parte de todos los miembros del equipo, sumada a las clases prácticas de shared memory y semáforos, se tuvo más claro qué hacer y por dónde comenzar.
- Hubo debates con respecto a cómo implementar el *shmADT*. Se planteó la idea de guardar dentro varios punteros, incluyendo uno de lectura y otro de escritura, entre otros. Pero finalmente se entendió alcanzaba con que cada instancia del ADT tuviera solo dos punteros: el de inicio de la zona de shared memory (devuelto por map) y otro *current*, ya sea de lectura o de escritura. Es decir, el puntero *current* es de lectura o de escritura según la instancia del ADT.
- Luego, durante el desarrollo, había un problema al momento de leer de shared memory. El proceso aplicación escribía bien en shared memory pero finalmente el vista recibía basura. Luego de analizar el código, se notó que en realidad había un problema al momento de definir el *newShm,* ya que se estaba haciendo un *sizeof* de *shmADT* en vez de *shmCDT*. Además también se había omitido llamar a *ftruncate* en el proceso vista. Finalmente se pudo solucionar.
- Para la liberación de recursos, había un memory leak del cual no se conocía su origen. El problema radica en que no se estaba liberando la instancia de shared memory en *masterADT* ya que no se podía acceder a dicho campo del ADT

directamente. Para solucionarlo, se creó una función en *shmADT* para liberar el espacio de memoria de la estructura.

- Un error importante cometido fue el de hacer *unlink* de la shared memory tanto en el *master* como en view. Esto ocasionaba un error ya que al intentar hacer el segundo *unlink* no encontraba un shared memory object con ese nombre (ya estaba desvinculado). Bastó con remover una de las dos llamadas para resolverlo.
- En cuanto a la sincronización de procesos, si bien se tenía la base teórica del uso de semáforos, nunca habíamos visto funciones concretas de la API de Linux. Bastó con leer el man para conseguir un punto de partida.
- Otro error fue hacer *wait* y *post* tanto en el write como en el read. Esto ocasionaba un problema de inanición, ya que el read es bloqueante por lo que nunca llegaba al post de la función.
- Finalmente, para pasarle al view cuántos archivos había que leer, se decidió pasarle el número en la primera posición de la shared memory como se mencionó previamente. El problema es que este pasaje era dinámico al momento de que el *master* iba leyendo cada archivo, y cuando llegaba a cero el vista cortaba antes de tiempo (los esclavos todavía estaban devolviendo resultados). Este problema se solucionó con una variable *filecount* en el master, que toma al iniciar el ADT el valor que se le pasa por parámetro, es decir es un valor fijo de archivos. Este valor sí se guarda en la primera posición de memoria de la shared memory.

Código de otras fuentes

En cuanto a fuentes externas, el código utilizado es completamente de autoría de los miembros del equipo. Solamente se utilizaron como referencia y ejemplo algunos códigos provistos por las páginas del manual de Linux, sobre todo de *shm_overview*, así como del ejemplo de shared memory dado en la clase práctica. Pero ninguno de ellos fue copiado de manera textual.