Trabajo Práctico N1

Comunicación Interproceso

06/04/2020

— Grupo 5

Martin Ciccioli
58361

Geronimo Maspero
58322

Matias Ricarte
58417

Ignacio Sampedro
58367

Introducción

El proyecto para el trabajo práctico en cuestión fue desarrollado utilizando el lenguaje de programación C, siguiendo las normas de un sistema POSIX. El objetivo del mismo es resolver múltiples fórmulas proposicionales en CNF de forma distribuida. Para ello, el sistema que hemos diseñado cuenta con 3 archivos diferentes: el **proceso aplicación**, los 5 **procesos esclavos** (hijos del proceso aplicación) y el **proceso vista**.

Decisiones Importantes

- Comunicación entre proceso Aplicación y proceso Esclavo

Para llevar a cabo la comunicación entre el proceso Aplicación y los procesos Esclavos, por cada par Padre-Hijo (es decir, Aplicación-Esclavo), hemos creado dos pipes, dado que estos son canales de datos unidireccionales.

Por un lado, el **pipe_application_slave** es utilizado para mandarle desde el proceso aplicación al proceso esclavo los paths relativos de los archivos que serán procesados por este último. Para ello, a través de la función dup2(), mapeamos la entrada estándar del proceso esclavo con el extremo de lectura de este pipe (pipe_application_slave). Dicho eso, cada vez que se mande un archivo a un proceso slave, el proceso aplicación usará la Syscall Write sobre el extremo de escritura del pipe_application_slave con el path del archivo seguido de un newLine. Luego, vimos cómo se usan ambos extremos del pipe_application_slave.

Por otro lado, el **pipe_slave_application** es el encargado de que la información procesada resultante del proceso esclavo pueda llegar al proceso aplicación. Para lograr esto, nuevamente a través de dup2(), mapeamos la salida estándar del proceso esclavo con el extremo de escritura de este pipe (pipe_slave_application). Dicho eso, cada vez que se se reciban datos viniendo desde un proceso Esclavo, el proceso Aplicación usará la Syscall Read sobre el extremo de lectura del pipe_slave_application. Nuevamente, vimos cómo se usan ambos extremos del pipe_slave_application.

Desde el proceso aplicación se espera a que haya datos para leer que fueron enviados por los procesos Esclavos, a través de los respectivos pipe_slave_application. Para leer dichos datos de la forma más eficiente, usamos la función select(), la cual le permite a nuestro programa monitorear múltiples File Descriptors, esperando hasta que al menos uno de ellos esté listo para ser leído y luego indicándonos cuáles de ellos están listos mediante FD_ISSET(). Luego de leer la información de los procesos Esclavos adecuados con la ayuda de select(), el proceso Aplicación verifica si aun hay archivos nuevos por procesar, y en dicho caso, se encarga de mandarle un nuevo archivo al proceso Esclavo que sabemos que está desocupado al momento.

- Comunicación entre proceso Aplicación y proceso Vista

Para la comunicación entre el proceso Vista y el proceso Aplicación usamos Shared Memory y semáforos. El proceso Vista puede recibir el nombre de la memoria compartida de dos formas distintas. Si el proceso Vista recibe el nombre por argumento significa que el Proceso Vista está separado del proceso padre, es decir, está corriendo desde otra terminal. En cambio, si lo recibe por entrada estándar (STDIN), tanto el proceso Vista como el proceso Aplicación están corriendo en la misma terminal. Dicho eso, es importante destacar que ambos procesos son independientes.

El proceso vista funciona de una forma simple. Una vez realizada la conexión con el proceso Aplicación, utiliza semáforos para esperar hasta que el proceso Aplicación haya escrito algo en la memoria compartida. Luego de que el proceso aplicación haga un sem_post, y espere con sem_wait a que Vista termine de leer de la memoria, el proceso Vista imprime lo que se halle en esta. Después, despierta al proceso Aplicación con un sem_post y se duerme hasta ser llamada de vuelta por este, con un sem_wait. Esto dentro de un ciclo hasta que el proceso Aplicación le mande un EOF, indicando que ya se imprimieron todos los archivos.

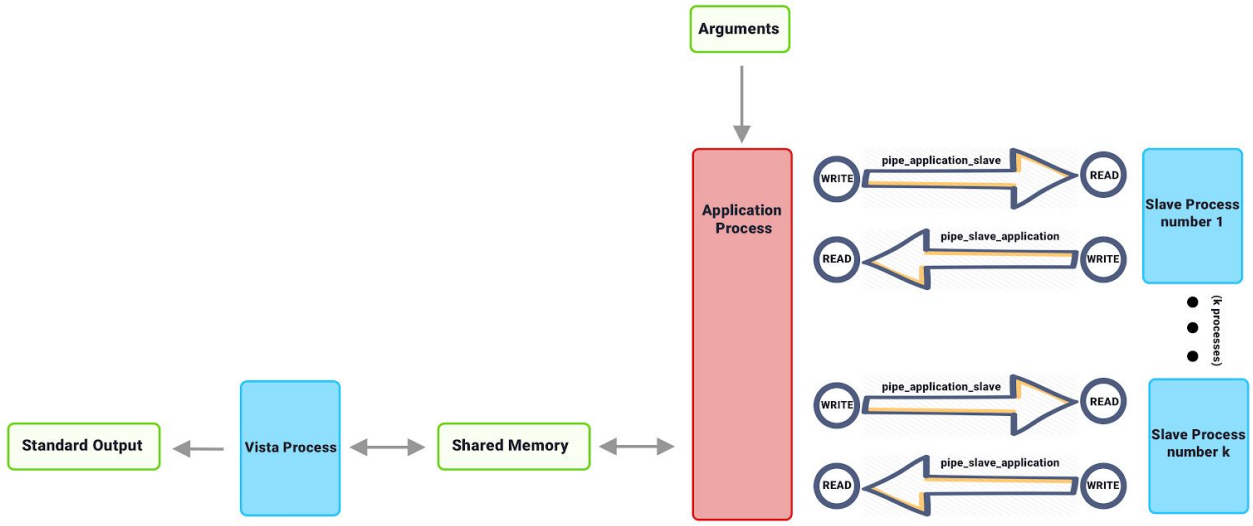
- ## Proceso Esclavo

Para lograr una distribución óptima sobre los archivos que deban ser procesados en nuestro programa, optamos por trabajar con 5 procesos esclavos diferentes en simultáneo. Adicionalmente, al comenzar el programa decidimos que cada proceso esclavo cuente con 2 archivos iniciales. Optamos por esta baja cantidad inicial para garantizar que nuestro mecanismo de distribución automático de archivos en demanda funcione correctamente.

Cuando el proceso Aplicación le pasa al proceso esclavo los paths de los archivos a procesar a través del pipe que ya vimos, este proceso usa la función getline() para recuperar esa información. Por este motivo, para asegurarnos que el proceso Esclavo lea un path por línea, desde el proceso Aplicación colocamos un salto de línea luego de enviar cualquier path al proceso Esclavo.

Una vez que el proceso Aplicación haya recibido todos los archivos procesados, este mismo le envía a cada proceso Esclavo un carácter especial de finalización, para avisarles que deben retornar. Por último, el proceso aplicación realiza un waitpid por cada uno de los procesos Esclavos para esperar su finalización.

Diagrama: cómo se relacionan los procesos



Instrucciones de compilación y ejecución

Se debe contar con minisat antes de seguir estos pasos. En caso de desear instalar minisat, se puede correr el siguiente comando en la terminal:

```
$ apt-get install minisat
```

Primero, para **compilar**, pararse en el directorio que tiene el código fuente de nuestro proyecto, y luego ejecutar el siguiente comando en la terminal:

```
$ make all
```

Para **ejecutar** el programa, hay tres posibilidades:

a) hacerlo desde una sola terminal y pipear la salida del proceso Aplicación al proceso Vista. Para ello, ejecutar en la terminal:

```
$  ./app <lista de archivos .cnf> | ./vista
```

b) hacerlo desde dos terminales separadas, ejecutando ambos programas por separado. Para ello:

1) Ejecutar en la terminal 1:

```
$  ./app <lista de archivos .cnf>
```

2) Ejecutar en la terminal 2:

```
$   ./vista <nombre de la memoria compartida>
```

c) hacerlo desde una sola terminal y solamente ejecutar el programa Aplicación. Para ello, ejecutar en la terminal:

```
$ ./app <lista de archivos .cnf>
```

Limitaciones

Dado que los esclavos reciben por un pipe archivos a procesar, y por otro retornan el resultado procesado de minisat, el proceso aplicación no tiene forma de saber cuántos archivos procesó en una respuesta. Para esto hubo que parsear la respuesta de cada esclavo y contar las veces que aparecía el substring “SATISFIABLE”. Esto agrega complejidad O(n) pero en las pruebas no hubo grandes diferencias notables. (Fuente 1)

Problemas encontrados durante el desarrollo

Para la comunicación entre el proceso Aplicación y el proceso Vista se utilizó una memoria compartida, donde ambos procesos escribirían y leerían datos. Para indicarle al proceso aplicación que el proceso vista está corriendo, ambos procesos realizan un “hand-shake” y continúan con su código. Un problema con esto es que vista no se conectaba correctamente al segmento de memoria compartida en el caso de correr ‘ ./app cnfs/* | ./vista’ en la terminal, generando un error. Para solucionarlo, después de imprimir el nombre usado para la memoria compartida en la Aplicación ejecutamos un fflush(stdout), forzando a imprimir todo en el buffer del stdout inmediatamente (Fuente 2), y desde la Vista nos aseguramos de que al leer el nombre, que haya un ‘\0’ al final de este.

Otro problema, no tan relevante pues se ha logrado solucionar fácilmente, fue también con la comunicación app <-> vista a través de semáforos. El problema era que se utilizaba solo un semáforo para el acceso a memoria, ya sea de lectura como de escritura. Se generaba una falla, ya que si uno quería acceder a la memoria y luego esperar a que el otro proceso accediera tenía que realizar un sem_post y luego un sem_wait. Claramente, hacer un sem_post y un sem_wait en la siguiente línea en cada programa generaba que, aleatoriamente, corriera estas 2 instrucciones seguidas sin pasar por el otro proceso (un sem_post(sem) y sem_wait(sem) cuando se corren juntos es como no hacer nada). El problema se solucionó generando 2 semáforos, llamados de lectura y escritura. El proceso Aplicación hace sem_post(sem_read) para informarle al proceso Vista que ya terminó de escribir datos, mientras que el proceso Vista hace sem_post(sem_write) para informarle al proceso Aplicación que ya terminó de leer datos.

Citas de fragmentos de código reutilizados

En este caso usamos KMP (visto en la materia Estructura de Datos y Algoritmos), el cual corre en O(n) en el peor caso, basandonos en la implementacion de la siguiente Fuente (1): https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/

Fuente (2): https://stackoverflow.com/questions/1716296/why-does-printf-not-flush-after-the-call-unless-a-newline-is-in-the-format-strin