Grupo 7

Trabajo Práctico Nro 1

Inter Process Communication

Integrantes:

Brula, Matías	*Legajo 58639*
Tallar, Julián	*Legajo 59356*
Vuoso, Julián	*Legajo 59347*

Introducción

Para la realización de este trabajo práctico, se realizaron tres archivos .c diferentes. El proceso **app** es aquel que crea los segmentos de memoria compartida, semáforos y pipes utilizados, crea y maneja los esclavos y realiza la escritura de la información en la memoria compartida y en un archivo. El proceso **slave** es aquel que lee el nombre de un archivo, ejecuta minisat sobre él y envía la información relevante resultante al proceso app. El proceso **vista** por su parte se encarga de leer la memoria compartida e imprimirla en pantalla. Los archivos .cnf de fórmulas a procesar los recibe el proceso aplicación por argumentos, son quienes serán procesados luego por minisat.

Decisiones tomadas

En primer lugar, decidimos trabajar con una cantidad de esclavos de SLAVE_NUMBER = 3 para poder observar una buena división de los archivos de prueba elegidos, evitando que alguno se quede sin archivos para procesar mucho tiempo antes que los demás. Respecto a la distribución inicial de archivos, decidimos repartir una cantidad de INIT_FILES = 3 a cada slave ya que preferíamos que fuera una cantidad poco significativa en relación al total de archivos para comprobar el correcto funcionamiento del pedido de archivos. Y las pruebas fueron realizadas con entre 40 y 60 archivos.

Para la comunicación entre el proceso app y el proceso vista, se utilizó una memoria compartida de nombre "/mem-*pid* ", donde *pid* es el process id del proceso app, que éste imprime por STDOUT al iniciar para que el proceso vista sepa su nombre. El tamaño asignado a la memoria compartida es de CHUNK_SIZE x FILE_NUMBER. Para sincronizar la lectura y escritura del buffer de la memoria compartida, se utilizó un semáforo entero con nombre de valor inicial 0. El proceso app escribe cada línea recibida en el buffer y realiza un post en el semáforo por cada cada una de ellas, mientras que el proceso vista realiza un wait en el semáforo antes de realizar la lectura de una línea.

Para la comunicación entre los procesos slave y el proceso app, se utilizaron pipes sin nombre unidireccionales: una por cada slave (app_to_slave[i]) para mandar los nombres de los archivos desde la app al slave y un único pipe (slave_to_app) para mandar la información desde los slaves a la app. Al crear un esclavo con fork y antes de ejecutar slave, se cierran los puertos STDIN y STDOUT y se duplican los file descriptors de los pipes creados. STDOUT pasa a ser un alias del extremo de escritura del pipe slave_to_app, mientras que STDIN pasa a ser un alias del extremo de lectura del pipe app_to_slave[i] correspondiente.

Cada vez que se deba enviar un archivo a un slave, el proceso app realiza un write sobre el pipe app_to_slave[i] correspondiente con el nombre del archivo seguido de un newLine. El proceso slave vinculado a ese pipe realizará la lectura de una línea (hasta un newLine) de STDIN (que está pipeado) para obtener el nombre y poder procesarlo.

Cada vez que se deba recibir un archivo desde un slave, el proceso slave realiza un write sobre STDOUT (que está pipeado) con la información completa del archivo procesado. Para controlar la escritura en el pipe slave_to_app de los distintos procesos slave, se utilizó un semáforo con nombre de valor inicial 1. Cada esclavo ejecuta un wait en el semáforo antes de escribir y luego realiza un post en el mismo. De esta manera, solamente un slave puede escribir al mismo tiempo, evitando el solapamiento de escrituras de distintos esclavos.

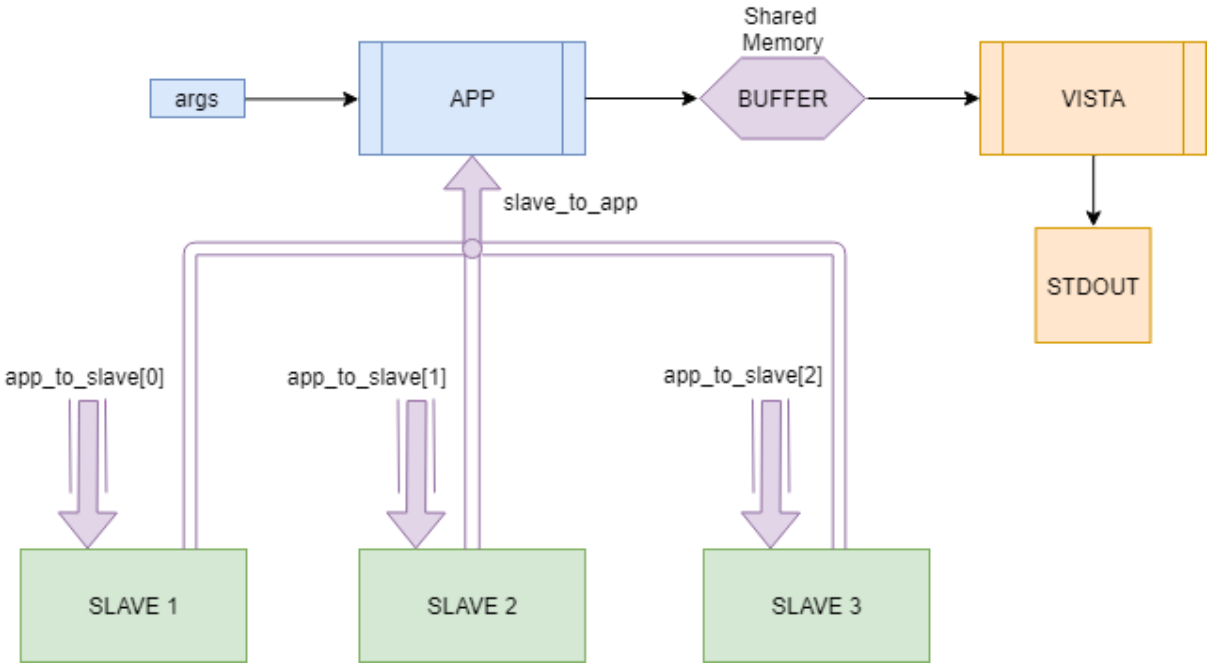
El proceso app realizará la lectura de slave_to_app línea por línea para obtener la información del archivo procesado. Una vez leídas 6 líneas (una por cada dato), se verifica si quedan archivos por procesar y si el slave asociado al identificador ya procesó todos los archivos enviados al mismo. De ser así, se le envía otro archivo.

Una vez que se han enviado todos los archivos y se han recibido las respuestas a cada uno de ellos, el proceso app le envía un caracter de finalización (' ? ') seguido de un newLine a cada uno de los procesos slave para avisarles que deben salir. También se imprime el mismo caracter en el buffer de la memoria compartida para indicar que no hay más información para leer. Por último el proceso app realiza un wait por cada uno de los esclavos creados, procediendo luego a su finalización.

Para ejecutar minisat y procesar su salida, utilizamos el comando popen, minisat y awk. La función popen permite ejecutar una línea de comando de la shell y leer su salida de un stream. De esta manera, podemos ejecutar minisat y tomar su salida como la entrada de otro comando de la shell.

Para parsear la salida de minisat, utilizamos el comando de linux awk. Este comando permite buscar patrones de un texto y tomar acciones según lo encontrado. En nuestro caso, buscamos las frases "Number of clauses", "Number of variables" y "CPU time" y "SAT" y tomamos solamente el número de palabra de la línea correspondiente a la información buscada, obteniendo los datos separados por un newLine. Luego se lee línea por línea del stream indicado al realizar popen y se guarda la información en un vector buffer.

Diagrama de procesos



Instrucciones de compilación y ejecución

Para chequear si minisat está instalado e instalarlo si no es así, ejecutar:

make minisat

Para **compilar** el programa, ejecutar:

make build o directamente make

Para **ejecutar** el programa, hay dos opciones: hacerlo desde una única terminal y pipear la salida de app a vista, o abrir dos terminales y ejecutar ambos programas por separado.

Si se quiere ejecutar todo en una única terminal, ejecutar:

./app Prueba/* | ./vista

Si se quiere ejecutar en dos terminales por separado, ejecutar:

Terminal 1: ./app Prueba/*

Terminal 2: ./vista

Se debe ingresar el PID impreso en la terminal 1 y presionar Enter

En ambos casos, Prueba/* refiere a los archivos a analizar, en este caso todos aquellos que se encuentran en la carpeta Prueba ubicada en el directorio actual.

También existe la posibilidad de ejecutar con la opción de terminal única tomando como argumento todos los archivos de la carpeta Prueba. Es equivalente a escribir ./app Prueba/* | ./vista. Para ello, ejecutar:

make run

Limitaciones

En primer lugar, consideramos que el tamaño máximo de cada bloque de información de un archivo procesado es de CHUNK_SIZE = 90. Este número se eligió tomando 38 bytes para el nombre, 7 para número de cláusulas, 7 para número de variables, 9 para el tiempo de procesamiento, 13 para el resultado, 4 para el id y 12 para los newLine y caracteres de finalización de string. Si el nombre del archivo es muy largo o el archivo tiene una gran cantidad de cláusulas, variables o requiere de mucho tiempo, el proceso slave fallará porque se reservaron CHUNK_SIZE bytes para la salida.

En segundo lugar, consideramos que la cantidad máxima de archivos ingresada por argumentos es de FILE_NUMBER = 600. El proceso app podría adaptarse a la cantidad de argumentos ingresados, pero como el proceso vista es independiente del proceso app y éste último solamente imprime por STDOUT lo necesario para conectarse a la memoria compartida, se debe fijar un máximo en el mapeo en memoria.

En tercer lugar, se fijó la cantidad de líneas por bloque en el valor CHUNK_LINES = 6 para que el proceso app sepa cuándo termina un bloque. Si se deseara imprimir más información desde cada proceso slave al proceso app, se debe modificar esa constante.

Problemas encontrados

Tuvimos inconvenientes en el parseo de la salida del minisat. Primero quisimos realizarlo en código, utilizando funciones como strstr y fscanf pero no pudimos lograr que funcione correctamente. Finalmente decidimos realizar el parseo utilizando el comando awk y ejecutarlo con popen.

También encontramos varios problemas con las funciones fprintf y fgets. Notamos un comportamiento incierto e imprevisible al utilizarlas, por lo que decidimos realizar toda lectura de puertos con getline.

Otro problema que surgió fue en la distribución de archivos desde el proceso app. En la salida del proceso vista, se observaba que los tres últimos bloques de información correspondían al mismo pid. Esto no era correcto ya que implicaría que otro proceso slave había devuelto su bloque y no habían recibido otro archivo aún habiendo otros por procesar. Notamos entonces que esto se debía la distribución inicial de archivos. Cada proceso esclavo pedía un archivo al terminar de leer uno, aún teniendo otros por resolver. Como al inicio distribuimos 3 archivos por esclavo, cada slave tenía 3 archivos dentro de su pipe constantemente, hasta que se terminaran los archivos. Para solucionarlo, agregamos una matriz para contabilizar la cantidad de archivos enviados y recibidos por cada esclavo, y solamente se le enviará uno nuevo cuando haya devuelto la información de todos los archivos que se le enviaron.

Respecto a los análisis estáticos y dinámicos, se corrigieron los errores encontrados en cada uno de los mismos. Hubo un único error que no consideramos tal, encontrado por PVS-Studio. Según este análisis, el proceso slave debería verificar el nombre del archivo recibido por STDIN (pipeada desde app) antes de ejecutar el popen del comando mencionado anteriormente. Sin embargo, los procesos slave reciben su información desde el proceso app y no desde STDIN, sumado a que consideramos que los archivos recibidos por argumento desde el proceso app son correctos. Por lo tanto, decidimos ignorarlo.

Por último, tuvimos inconvenientes en el unlink de la memoria compartida y el semáforo de la misma. Cuando hacíamos ./app Prueba/* | ./vista, el proceso vista tiraba error porque no podía realizar la apertura de lo mencionado, como si ya se hubiera ejecutado el unlink de los mismos. Para solucionarlo, realizamos un fflush(stdout), forzando la ejecución del printf apenas se lo escribe.

Citas de fragmentos de código reutilizados

Para la lectura de información línea por línea, nos basamos en el siguiente fragmento de código del Ejercicio 5 del TP0 de la materia:

```
int main(int argc, char *argv[]) {
    char *line = NULL;
    size_t linecap = 0;
    ssize_t linelen;
    int linecount = 0;

    while ((linelen = getline(&line, &linecap, stdin)) > 0) {
        linecount++;
        printf("\nLine %d\n", linecount);
        fwrite(line, linelen, 1, stdout);
    }
    return 0;
}
```