ITBA
Instituto Tecnológico de Buenos Aires

Trabajo Práctico N° 1
Inter Process Communication

Fecha de entrega: 18/04/2021

Materia: Sistemas Operativos

Cátedra:

- Aquili, Alejo Ezequiel
- Godio, Ariel
- Merovich, Horacio Víctor
- Mogni, Guido Matías

GRUPO 8:

- Hinojo Toré, Nicole; 57440
- Larroude Álvarez, Santiago Andrés; 60460
- Menghini, Mateo; 60090

Decisiones de desarrollo

Se tomó el ejemplo de la consigna como propuesta respecto a la cantidad de procesos a crear: se decidió utilizar 5 procesos *workers* que inicialmente reciben 2 *files* para resolver.

Con respecto a la comunicación entre el proceso *master* o *application* con sus *workers* utilizamos **dos *unnamed pipes***:

- **`fd_works`**
  Envía los archivos a los *workers* para que ellos lo/los resuelvan.

- **`fd_results`**
  Los *workers* envían las soluciones de los archivos para que el padre procese.

Esto es posible ya que hay un único proceso *master* que crea a todos los hijos, los *workers*, que se ejecutan independientemente de la *view*.

Luego, para la comunicación entre el master y el proceso *view* se creó un **espacio de memoria compartida con un semáforo**. De esta manera, se coordinan los accesos de lectura y escritura, para evitar condiciones de carrera y *deadlocks*.

La implementación de la shared memory está diseñada de tal forma que el escritor siempre escribe en posiciones de memoria diferentes, es decir, no sobreescribe información. Por lo tanto, la memoria tiene un tamaño acorde para que todos los resultados estén guardados a la vez, y están separados a través de un '\0'. El lector realiza la lectura a medida que va recibiendo información, y al estar cada resultado separado por un '\0' se los puede diferenciar fácilmente y saber cuántos resultados ha leído.

El semáforo representa cuántos resultados hay en la memoria para ser leídos (está inicializado en 0); por lo tanto la vista se bloquea hasta que aparezca un resultado para leer (no hay busy waiting, ni tampoco condiciones de carrera, ya que solamente lee cuando hay un resultado), y el master incrementa el semáforo cuando termina de escribir un resultado. Esto es posible debido a la implementación de la shared memory donde el master escribe en posiciones diferentes cada vez que lo hace y view lee en posiciones que no serán modificadas luego. Se asegura la correcta integridad de la lectura a través del semáforo, es decir, view no leerá un resultado hasta que este esté completamente guardado y el semáforo haya realizado el incremento correspondiente.

El proceso *application* vuelca los resultados en un archivo con el nombre ***output_application***. En caso de que no exista, el mismo lo crea; caso contrario, guarda los nuevos resultados, dejando un historial de resoluciones.

Además, los procesos *workers* **reciben** las tareas a resolver vía **entrada estándar** y **devuelven** sus resultados vía **salida estándar** desde y hacia el proceso *application* respectivamente, de manera que se pueda ejecutar dicho proceso de manera aislada en la terminal sin necesidad de que el proceso a*pplication* lo cree.

Diagrama de procesos

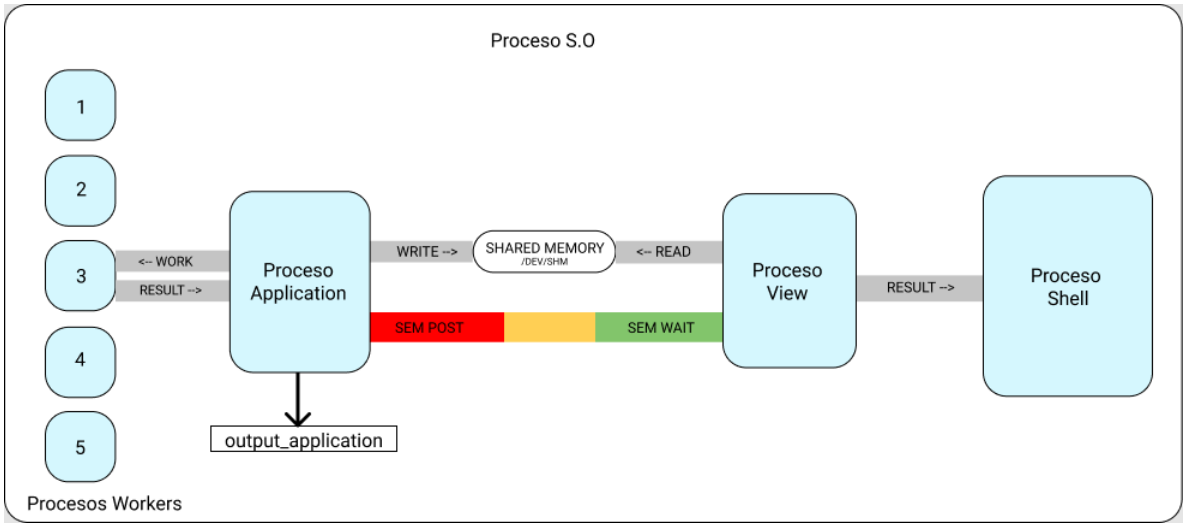

Figura 1: Diagrama de Procesos

En el diagrama se puede observar tanto la comunicación entre application y los workers (aplica para todos los workers de la misma forma), como la de *application* con view. La distribución de tareas se realiza a través de *pipe works*, mientras que el envío de las respuestas se envía a través de pipe results. Ambos pipes son exclusivos de cada *worker*, por lo que no se producen condiciones de carreras entre ellos a la hora de retornar las respuestas. La memoria compartida es apuntada por *application* para escribir la información correspondiente a cada tarea finalizada, y es apuntada por la view para que pueda leerla e imprimirla por salida estándar. Esta memoria es sincronizada a través de un semáforo. Además, el proceso application realiza el volcado de la información sobre un archivo *output_application*.

Instrucciones de instalación y ejecución

Dentro del entregable se puede ver un archivo Makefile, donde que el usuario deberá situarse en el directorio del mismo para luego correr los comandos `make clean` y `make all`. Al ejecutar este último, se crearán 3 archivos: **application**, **worker** y **view**; los cuales se pueden ejecutar de 3 formas:

1. `./application` *`path`*

Se ejecutará el proceso padre enviando los archivos dentro de *path*. No se mostrará ningún resultado por consola, en cambio estos resultados se podrán ver dentro del archivo ***output_application*** como se mencionó anteriormente ejecutando el comando `cat output_application` lo veremos en consola.

2. `./application` *`path`* `| ./view`

   Realiza lo mismo que en el caso anterior (ejecutar el proceso padre) y ejecutando el proceso **view**, que se conectará a un espacio de memoria compartida para leer los resultados del proceso **application** y mostrarlos por consola a medida que se resuelvan los archivos.

3. `./worker [ENTER]`
   `file` `[CTRL+D]`

   Ejecuta el proceso hijo/esclavo y procesa este *file/s* que se envíe/n dentro de la ejecución. El `[CTRL+D]` corresponde al *EOF* de la *shell* para terminar la ejecución del proceso.

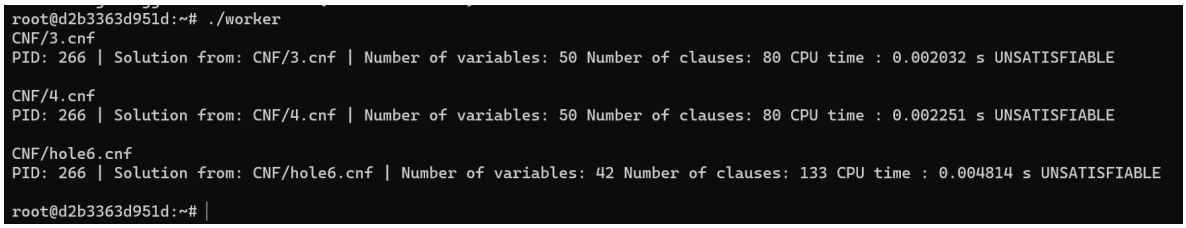

Figura 2: Ejemplo de ejecución del comando 3.

Limitaciones

Por decisiones de implementaciones, el proceso raíz tiene 5 procesos hijos los cuales enviaran 2 archivos a cada uno en una primera instancia. Por lo tanto, **tomamos como precondición que se pasan al menos 10 archivos para procesar.**

Una limitación del programa es que la *shared memory* necesita tener disponible una cantidad de memoria tal que todos los resultados puedan estar guardados a la vez. Si bien es ineficiente guardar la información de esta manera, con las capacidades de memoria actuales de las computadoras, no representa un problema en la gran mayoría de usos. Se realiza esta implementación para facilitar la relación entre el proceso application y el proceso view. Otra limitación es que cuando el programa falla se realiza un `exit` instantáneamente, por lo tanto, pueden haber recursos como la shared memory y el semáforo que no fueron cerrados correctamente.

Problemas durante el trabajo

- Al momento de testear la implementación de la memoria compartida, se decidió utilizar `shm_unlink` para evitar **ENOENT**.
- Un problema que no pudimos resolver por cuestiones de tiempo, es que el *Valgrind* a veces funciona en corto tiempo y a veces no. El patrón descubierto es que sucede en los casos que volvemos a ejecutar *application* sin hacer un `make clean` antes.
- Al momento de comunicar los procesos Application y Worker, la información que contenía los resultados de cada tarea no llegaba a application, por lo tanto éste no los podía procesar, es decir se quedaban bloqueados en un buffer. Para desactivar el buffer se agregó la sección de código **setvbuf(stdout, NULL, _IONBF, 0)** en el proceso worker.
- Por la ausencia de la función `setvbuf` en el proceso *application*, el tamaño de la memoria compartida que debía ser impreso al momento de la ejecución de *application*, se visualizaba por **STDOUT** previo a la finalización de la ejecución del programa. Gracias a la adición de la línea de código anteriormente mencionada (setvbuf) se solucionó el problema en cuestión.

Códigos fuente

- Ejemplo de uso de Shared Memory:
  - https://github.com/WhileTrueThenDream/ExamplesCLinuxUserSpace
- Parseo de parámetros por entrada y salida estándar usando `popen` y `fgets`:
  - https://pubs.opengroup.org/onlinepubs/009696799/functions/popen.html
- Comando `tar` para eliminar espacios en blanco del parseo de `grep`:
  - https://stackoverflow.com/questions/12763944/shell-removing-tabs-spaces