Sistemas Operativos

Primer cuatrimestre 2021

Trabajo Práctico Nro. 1: Inter Process Communication

Integrantes:

- Arce Doncella, Julián Francisco - 60509.
- Domingues, Paula Andrea - 60148.
- Lombardi, Matías Federico - 60527.

Instrucciones de Compilación

Ubicado dentro del directorio donde se descargaron los archivos, ejecutar el comando:

```
$ make all
```

Esto generará todos los archivos ejecutables necesarios.

Instrucciones de Ejecución

El programa master guardará todos los resultados en un archivo, sin necesidad de otro proceso, para ello, ejecute:

```
$ ./master ./Files/*
```

Adicionalmente, se puede ejecutar el proceso view, que se encargará de imprimir por salida estándar los resultados, de la siguiente manera:

```
$ ./view <MEMORY_SIZE>
```

donde el valor <MEMORY_SIZE> será indicado por salida estándar al ejecutar el proceso master. Para ello, se cuenta con 2 segundos de espera.

Para iniciar ambos procesos a la vez, ejecutar lo siguiente:

```
$ ./master ./Files/* | ./view
```

Instrucciones de Testeo y Limpieza

Finalmente, en el caso de querer realizar los testeos utilizando Valgrind, cppcheck y pvsStudio Analyzer basta con ejecutar lo siguiente, siempre y cuando nos encontremos en un entorno con los respectivas programas instalados

```
$ make test
```

Lo cual generará tres archivos de lectura, uno para cada rutina de testeos ejecutada, master.valgrind el cual contará con el output de Valgrind, cppoutput.txt correspondiente al cppcheck y report.tasks, el cual corresponde a pvs-studio.

Adicionalmente, en caso que se quisieran eliminar estos archivos se dispone de las siguientes opciones:

```
$ make clean
```

para eliminar los tres ejecutables creados por el make all,

```
$ make clean_test
```

para eliminar los archivos de texto creados por el make test, y por último

```
$ make clean_all
```

la cual engloba los dos clean anteriormente mencionados.

Decisiones tomadas durante el desarrollo

Creación de *Slaves*

Los procesos *slaves* se crean a través de la siguiente política:

- Se crean un máximo de 5 *slaves*.
- Si la cantidad de tareas es menor a 5, se crean tantos *slaves* como tareas.
- Si la cantidad de tareas es 10 o más, al crear cada *slave*, se le asignan 2 tareas a cada uno.
- Si la cantidad de tareas es menor a 10, se le asigna 1 tarea a cada *slave* al ser creado.
- En caso de haber más tareas para procesar, las mismas son asignadas por demanda, es decir, si un *slave* se libera, recibe una tarea más para completar.
- Si ya no hay más tareas para asignar, los *slaves* finalizan su ejecución cuando terminan de procesar todas sus tareas ya asignadas.

Se asignan tareas a los *slaves* al ser creado para que, aunque por un tiempo insignificante, no estén ociosos y empiecen a procesar archivos inmediatamente. Estos valores pueden ser cambiados en sus respectivas constantes y el programa seguirá funcionando correctamente.

Comunicación entre Procesos

Para la comunicación entre el proceso *master* y los procesos *slaves*, se utilizan dos pipes unidireccionales por *slave*. En uno de los pipes (*pipe path*) se realiza la comunicación desde el *master* al *slave* y se transmiten los *paths* de los archivos que deben ser procesados en *minisat*. El otro pipe (*pipe answer*) se conecta desde el *slave* al *master* para enviar los resultados de cada archivo procesado por *minisat*.

Al tener un *pipe answer* por *slave* no se producen condiciones de carrera debido a que cada *slave* escribe sobre un *pipe* diferente y además podemos saber qué proceso es el que produjo dicha respuesta para luego asignarle más tareas a través de su *pipe path* correspondiente o retirar su *fd* correspondiente del set que se utiliza para recibir las respuestas, en caso de que no hayan más tareas para asignar. El *master* detecta que recibió un resultado debido a que se activa la función *select*. De esta manera, se puede esperar la respuesta de varios *slaves*, en vez de ejecutar un *read* por cada uno (lo que sería muy ineficiente, ya que mientras se está esperando la respuesta de un *slave*, se podría estar procesando el resultado de otro). Con esta implementación no se necesita que las operaciones de escritura sobre *pipe answer* sean atómicas entre los *slaves* ya que escriben sobre secciones diferentes, y luego el *master* lee cuando se activa el *select*, como se mencionó anteriormente.

Por otro lado, el proceso *master* y el proceso *view* comparten información a través de un segmento de memoria a la cual pueden acceder ambos procesos. El proceso *master* cumple el rol de escritor y el proceso *view*, el de lector.

Investigando el problema de escritores y lectores, otra posible implementación que permite realizar la comunicación y la sincronización de los procesos es a través de *Cola de Mensajes*, las cuales permiten la comunicación asíncrona, es decir, los puntos de conexión

que producen y consumen mensajes interactúan con la cola, no entre sí. Esta opción podría ser una futura alternativa al uso de pipes.

Implementación de la Shared Memory

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.

Mecanismo de Sincronización

Para sincronizar la escritura y lectura de la *shared memory* se utiliza un *semáforo*. Para esto, se evaluaron dos opciones diferentes.

La primera era inicializar el *semáforo* en 1, por lo que un solo proceso (entre *view* y *master*) podía acceder a la memoria, mientras que el otro se bloqueaba hasta que se libere; esta opción fue rechazada porque no es lo más óptimo que el master se encuentre bloqueado cuando quiere escribir en la memoria, ya que se desperdicia tiempo de procesamiento cuando llegan resultados.

La segunda posible semántica, es que el semáforo represente 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 no se cuenta con el problema mencionado en la primera opción, porque el *master* simplemente 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.

Teniendo en cuenta las ventajas y desventajas de cada opción, la segunda semántica fue la elegida debido a que facilita la implementación del semáforo.

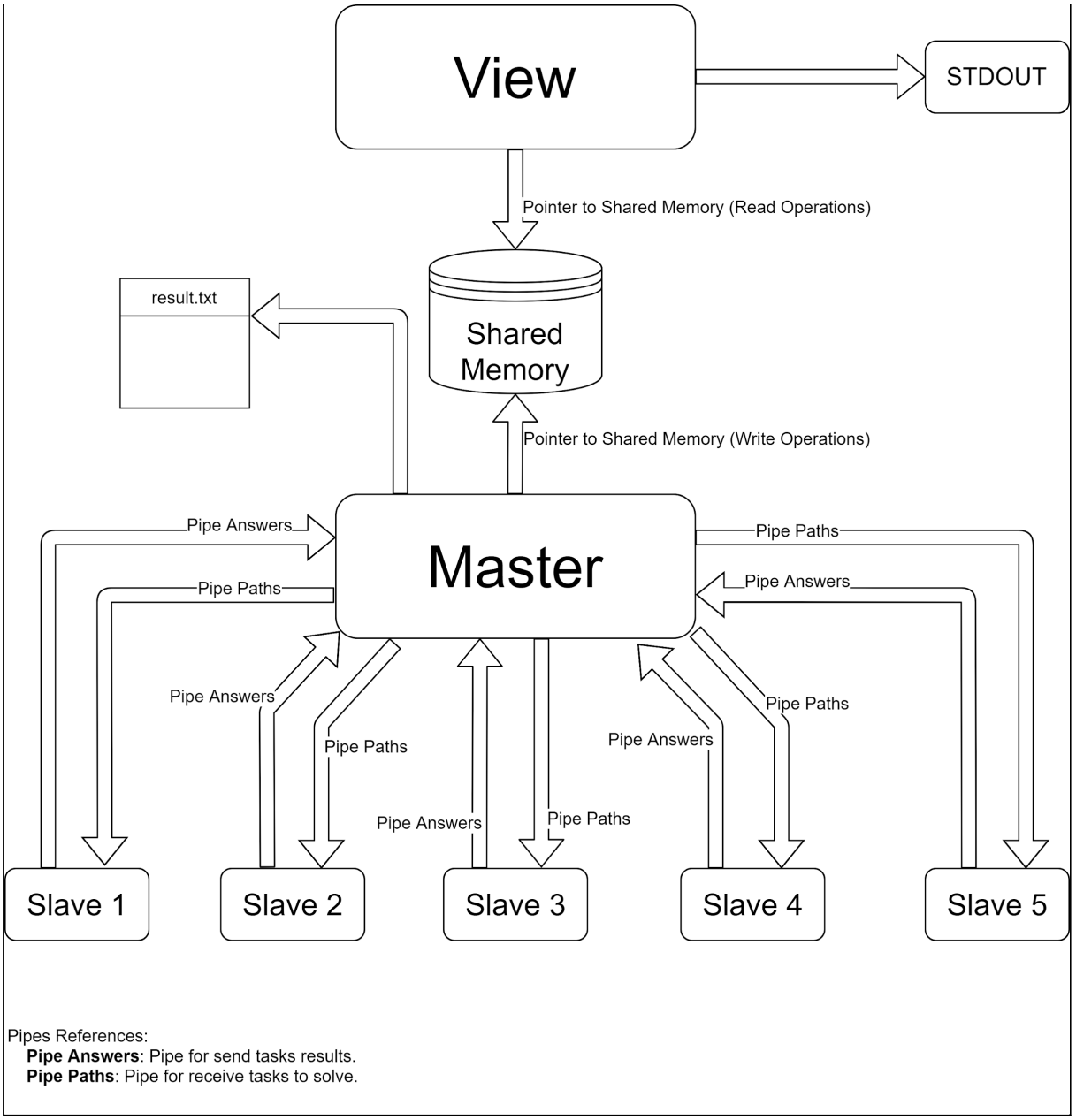
Manejo de Errores

El manejo de errores está implementado a través de una macro definida en el manual, que permite el pasaje de un mensaje. Realiza la interpretación de *errno*, imprime el mensaje correspondiente y luego finaliza la ejecución del programa.

Liberación de los recursos de *Slaves*

Se decidió esperar a que se procesen todas las tareas para cerrar los *pipes* que corresponden a las respuestas, al mismo tiempo que se hace un *wait* por cada uno. Si bien esto se podría realizar al retornar *EOF* por este *pipe*, se concluyó que lo más simple y práctico era dejar de "escuchar" los *fd* al procesar todas las tareas, y hacer la liberación de estos recursos antes de terminar la ejecución del *master*, respetando la *API.*

Diagrama de Comunicación de Procesos



En el diagrama se puede observar tanto la comunicación entre en el *master* y los *slaves*, como la de *master* con *view*.

La distribución de tareas se realiza a través de *pipe paths*, mientras que el envío de las respuestas se envía a través de *pipe answers*. Ambos *pipes* son exclusivos de cada esclavo, por lo que no se producen condiciones de carreras entre *slaves* a la hora de retornar las respuestas.

La memoria compartida es apuntada por el *master* para escribir la información correspondiente a cada tarea finalizada, y es apuntada por la *view* para que ésta pueda leerla e imprimirla por la salida estándar. Ésta memoria es accedida a través de un semáforo.

Además, el *master* realiza el volcado de la información sobre un archivo de texto, *result.txt*.

Limitaciones

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 actual de las computadoras, no representa un problema en la gran mayoría de usos. Se realiza esta implementación para facilitar la relación entre el proceso *master* 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 encontrados durante el desarrollo y sus soluciones

La mayor problemática que surgió durante el desarrollo del proyecto fue al momento de comunicar los procesos *Slave* y *Master*. La información que contenía los resultados de cada tarea no llegaba al *master*, por lo tanto éste no los podía procesar, es decir se quedaban bloqueados en un buffer. Para desactivar el buffer se agregó la siguiente sección de código `setvbuf(stdout, NULL, _IONBF, 0)` en el *slave*.

También, por esta misma razón y la ausencia del setvbuf en *master.c*, el tamaño de la memoria compartida que debía ser impreso al momento de la ejecución del *./master,* se visualizaba por STDOUT en el instante previo a que finalice la ejecución del programa. Nuevamente la adición de la línea de código anteriormente mencionada solucionó el problema en cuestión.

Otro problema se dio a la hora de crear el semáforo con *sem_open*, ya que para crear un semáforo se debe usar la llamada con 4 parámetros y no la que utiliza 2 parámetros, la cual sólo abre un semáforo, pero no lo crea. Esta indicación está dada por el manual.

Al momento de realizar los respectivos testeos con strace, en caso de no deshabilitar el buffering (*setvbuf*) del archivo *results.txt*, los resultados no se imprimen a medida que llegaban, sino que se hace un único *write* en caso de que el buffer se llenara, y al momento de realizar un *close* (según el manual *"The fclose() function flushes the stream pointed to by stream (writing any buffered output data using fflush(3)) and closes the underlying file descriptor."*). Esto generó que la salida de strace sea la siguiente:

```
munmap(0×7f1852539000, 225280)          = 0
unlink("/dev/shm/shared")               = 0
munmap(0×7f18527df000, 32)              = 0
unlink("/dev/shm/sem.sem")              = 0
write(3, "             : 0.00264 s\nSATISF"... , 4016) = 4016
close(3)                                = 0
wait4(-1, NULL, 0, NULL)                = 39575
wait4(-1, NULL, 0, NULL)                = 39576
wait4(-1, NULL, 0, NULL)                = 39578
wait4(-1, NULL, 0, NULL)                = 39581
wait4(-1, NULL, 0, NULL)                = 39582
exit_group(0)                           = ?
+++ exited with 0 +++
```

en donde se observa que se realizan los *unlinks* tanto de la memoria compartida como de los *semáforos* utilizados y luego el *write* (que se debe al *fclose* previamente mencionado).

Al deshabilitar el *buffering*, no hay un *write* final al realizar el *fclose*, sino que se realiza uno por cada resultado:

```
munmap(0×7fbb737f0000, 225280)          = 0
unlink("/dev/shm/shared")               = 0
--- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_EXITED, si_pid=38377,
munmap(0×7fbb73a96000, 32)              = 0
unlink("/dev/shm/sem.sem")              = 0
close(3)                                = 0
wait4(-1, NULL, 0, NULL)                = 38377
wait4(-1, NULL, 0, NULL)                = 38378
wait4(-1, NULL, 0, NULL)                = 38379
wait4(-1, NULL, 0, NULL)                = 38382
wait4(-1, NULL, 0, NULL)                = 38384
exit_group(0)                           = ?
+++ exited with 0 +++
```

Como estas operaciones no están involucradas en la sincronización de procesos, no representa un problema el uso de *buffering*, por lo tanto, su uso no afecta al programa.

Citas de fragmento de código reutilizados de otras fuentes

Se utilizó un fragmento de código externo al momento de parsear las respuestas devueltas por los esclavos al máster a través de un pipe, utilizando la función *strtok* de la librería "*string.h*". Se utilizó el siguiente código como base de cómo utilizar el mismo:

```
int main(){
    char cadena[] = "textoA, textoB, textoC. TextoD",
    delimitador[] = ",. ";
    char *token = strtok(cadena, delimitador);
    if(token != NULL){
        while(token != NULL){
            printf("Token: %s\n", token);
            token = strtok(NULL, delimitador);
        }
    }
}
```

Strtok

Para el manejo de errores, se utilizó una macro provista por el manual de Linux:

```
#define handle_error(msg) do { perror(msg); exit(EXIT_FAILURE); } while (0)
```

Para el uso de memoria compartida, además de consultar el manual, también se consultó la página POSIX shared-memory API - GeeksforGeeks.

Para calcular el máximo al momento de crear el *set* de *fd* para el uso de la función *select,* se utilizó la siguiente función que se encontraba en un ejemplo del manual de Linux:

```
#define max(x,y) ((x) > (y) ? (x) : (y))
```