Sistemas Operativos
Trabajo Practico N°1: Inter Process Communication

Grupo 5:

Alberto Abancens 62581
Gonzalo Rossin 60135
Uriel Mihura 59039

13/9/2021

1. Introducción

En este informe se estará presentando cómo se llevó a cabo el TP1 de la materia Sistemas Operativos.

En escencia, el programa final ejecuta y resuelve varios problemas, conocidos como *SAT solving*, utilizando el conocido programa *minisat*. Se utilizó éste ya que el trabajo está en hacerlo correr varias veces en paralelo, pero que al final del día no importe si se utiliza el minisat, o cualquier otro programa que se desee correr en paralelo. Esto es para así hacer uso de los multiples *cores* de los procesadores modernos, utilizandolos para resolver varios problemas con mayor velocidad que si se estuviesen resolviendo uno detrás de otro en el mismo núcleo.

Al inicializar el programa, éste toma por argumento los archivos .cnf que deberán ser resueltos. Luego se crean varios (discutido mas adelante) procesos *slaves*, o esclavos, de los cuales cada uno se encargará de ir resolviendo (de a una) las tareas que se le asignen. Estos procesos esclavos estarán corriendo a la par, y todos se comunicarán con el *master* via *pipes* (también discutido mas adelante), que son inicializados al crear cada proceso esclavo. Luego, siempre y cuando no se hayan resuelto todos los problemas, el *master* le irá asignando un problema a resolver a cada *slave*, el *slave* terminará de resolverlo y el *master* le asignará otro.

A medida que se vayan resolviendo los archivos, otro proceso llamado el *view* (que debera ser ejecutado en otra terminal) irá imprimiendo en pantalla los datos relevantes, que tomará desde una memoria compartida que comparte con el proceso *master* (concepto de *shared memory*, también discutido mas adelante).

2. Decisiones tomadas

Para el desarrollo de este programa, dado que la principal problemática era la implementación de una eficaz IPC, se utilizaron *pipes* y *shared memory*. Los pipes utilizados fueron *unnamed pipes*, todos anónimos y unidireccionales, ya que se decidió que los *named pipes* solo sumaban a la complejidad del código y no por esto iba a necesariamente quedar mejor armado.

Se decidió que la cantidad de esclavos inicializados sean 5, a menos que la cantidad de tareas sea menor a 5 en cual caso la cantidad de esclavos a inicializar será la misma que la cantidad de tareas a resolver. Además, a cada esclavo se le asignan 2 pipes (o mejor dicho un extremo de cada uno de estos 2 pipes), de los cuales en el otro extremo estará el master, ya sea leyendo datos en uno de los pipes o escribiendo datos en el otro. Se decidió este formato de pipes ya que utilizar un unico pipe para ambas direcciones de flujo de información trae enormes dificultades a la hora de implementarse.

En cuanto al uso de la shared memory y los semaforos, se conformó una API propia las cuales son wrapers de funciones para crear y unirse a la shared memory así como para el manejo del semaforo. Esto simplifica el uso de estas funciones dentro del código de *Master* y *View*, como también hace mas simple el manejo de errores en nuestra API que lo que resultaría del uso de estas funciones de biblioteca estandar de C.

Por otra parte, para el diseño de la shared memory se decidió que la cantidad de espacio reservada fuera en función de la cantidad de tareas a realizar al momento de ejecutar el programa. De esta forma, no se reservan cantidades muy grandes de memoria cuando las tareas a realizar son pocas, ni tampoco se da el caso de no haber suficiente espacio cuando la cantidad de tareas a realizar es muy grande.

Por último, tanto para el manejo de la shared memory como del semáforo, se elaboraron estructuras cuya función es la de guardar información relevante (*file descriptors*, *punteros*, indices de lectura y escritura, etc). Se notó que esto sería muy útil para la administración de la memoria y el semáforo, y aún mas importante para su posterior cierre al finalizar las tareas.

3. Diagrama

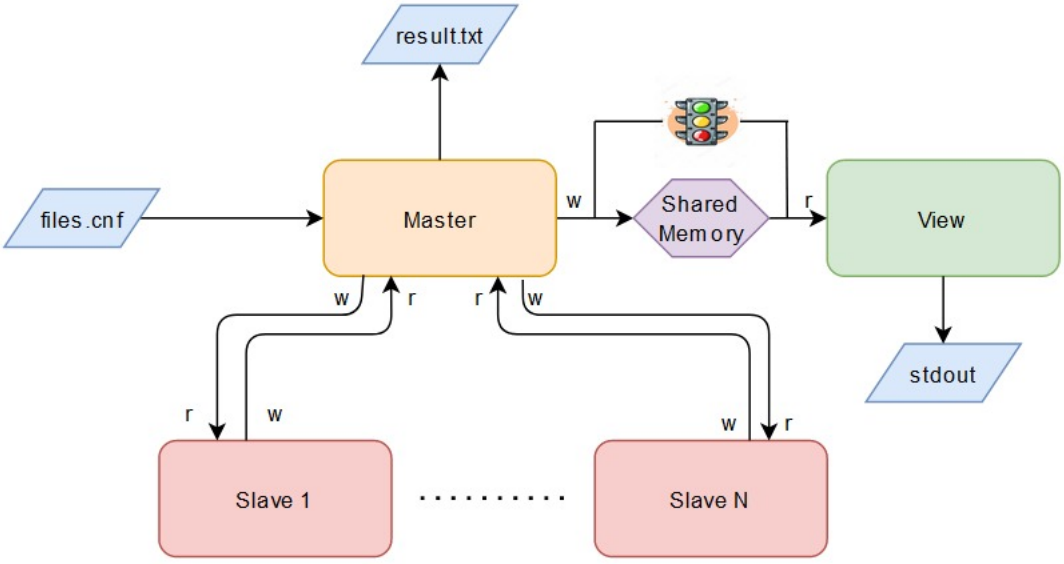

Figura 1: Diagrama del proyecto

4. Instrucciones de compilación y ejecución

Desde la carpeta 'TP-1-SO', se puede ejecutar el comando *make all*, este ejecutará los contenidos del archivo *Makefile*, el cual se encargará de compilar el proyecto:

```
gcc -Wall -c master.c -lrt -pthread
gcc -Wall -c shared_memory.c -lrt -pthread
gcc -Wall master.o shared_memory.o -o Master -lrt -pthread
gcc -Wall -c slave.c -lrt -pthread
gcc -Wall slave.o -o Slave -lrt -pthread
gcc -Wall -c view.c -lrt -pthread
gcc -Wall view.o shared_memory.o -o View -lrt -pthread
```

Luego, para ejecutar el programa se deberá correr el programa *Master*, pasandole como argumento los archivos .cnf a ser resueltos, en este caso en particular se han guardado algunos archivos ejemplo dentro de la carpeta *files*, entonces se puede correr lo siguiente:

```
./Master ./files/*
```

El programa se ejecutará, inicialmente imprimiendo en pantalla la cantidad de argumentos que este leyó correctamente. Una vez inicializado, se podrá correr el programa de vista (en otra terminal) para visualizar las salidas de los procesos esclavo. Para ejecutarlo correctamente se le deberá pasar por parametro el número recién impreso por *Master*:

./View 4

Alternativamente, se puede ejecutar todo en un solo comando, usando un pipe común "—"para redireccionar la salida de *Master* a *View*:

./Master ./files/ — ./View*

5. Limitaciones

Se asume que minisat no arroja ningún mensaje de error al momento de ejecutar los archivos .cnf ya que creemos que la validación de estos archivos no entra en los conceptos prácticos ni teóricos de la materia.

El tamaño de los pipes es un valor fijo (4096 bytes), que arrojaría un error de *buffer overflow* si se tratase de comunicar un string de mayor tamaño. De todos modos, es un valor 'estándar' y se cree que no debería suceder este error ya que es un valor bastante grande para lo que se está usando.

Actualmente se generan como mucho 5 procesos esclavos para resolver los ejercicios, lo cual podría ser poco práctico en el caso de querer resolver demasiados archivos. Este valor puede ser editado cambiando el valor de la macro *SLAVE_INIT* del archivo *master.c*.

6. Problemas Encontrados y sus Soluciones

A la hora de implementar el *select* en el master, se encontraron algunas adversidades. Luego de ser enviado *EOF* desde el read, el *select* seguía seleccionando al mismo file descriptor. Este problema fue resuelto evitando entrar en la seccion de procesamiento de tareas cuando el read devuelve 0.

La implementación de los semaforos también fue un dilema, comenzamos utilizando semaforos sin nombre pero no pudimos resolver que ambos procesos utilicen el mismo semaforo satisfactoriamente. En vista de esto decidimos utilizar semaforos con nombre para que cada proceso pueda acceder a el con mas naturalidad.

El proceso esclavo no estaba comunicandose apropiadamente con el master, no devolvía al pipe sus prints hasta que finalizace el proceso. Para solucionar

eso se eliminó el buffer de su salida estandar con la función *setvbuf*. Luego implementar la comunicación del *Master* con *View* en caso de *pipear*los fue mas intuitiva puesto que ya teniamos este insight.

Durante el desarrollo, ocasionalmente ocurría un bug en el que se imprimían algunos caracteres extra, o también *muy* ocasionalmente el *shm_open* devolvía -1.

El *stdout* generaba problemas con su buffer al enviar su respuesta. Para arreglarlo se le implementanó *setvbuf* en las 3 partes del programa, *Master*, *Slave* y *View*.

7. Bibliografía

Para este trabajo se requirio el uso intensivo del manual para el uso de las funciones de biblioteca estandar y el manejo de errores de las mismas. Por otra parte, para los primeros pasos del tp, recurrimos a una serie de videos dedicados a la comunicación entre procesos del canal de youtube çodevault". Por ultimo, para el desarrollo del tp consultamos el siguiente repositorio:

https://github.com/Reversive/ipc-sat-solver