Sistemas Operativos

Trabajo Práctico Especial Nº1

Alvaro Crespo	50758	acrespo@alu.itba.edu.ar
Juan Pablo Civile	50453	jcivile@alu.itba.edu.ar
Darío Susnisky	50592	dsusnisk@alu.itba.edu.ar

12 de Septiembre del 2011

Índice

1. Introducción

El objetivo de este trabajo es familiarizarse con el uso de sistemas cliente-servidor concurrentes, implementando el servidor mediante la creación de procesos hijos utilizando *fork()* y mediante la creación de *threads*. Al mismo tiempo, ejercitar el uso de los distintos tipos de primitivas de sincronización y comunicación de procesos (IPC) y manejar con autoridad el *filesystem* de Linux desde el lado usuario.

2. Esquema de la aplicación

A la hora de encarar el problema en cuestión, una de las primeras cosas a plantearse era el diseño de la aplicación. Dada la naturaleza y los objetivos planteados para este trabajo práctico, era necesario poder identificar que partes de nuestra simulación iban a ser procesos paralelos y cuales tenía coherencia implementarlas como *threads*. En este debate, también fue importante no forzar la separación de procesos cuando el problema no lo requería (por ejemplo que cada avión fuera un proceso independiente no agregaba nada y generaba mucha complejidad, por lo que se optó por implementarlos como *threads*).

En un primer análisis, identificamos como potenciales procesos paralelos de nuestro programa al control del flujo principal del programa, a los parsers, al mapa, al *output*, a las aerolineas y a los aviones. A continuación se puede ver un esquema de este modelo.

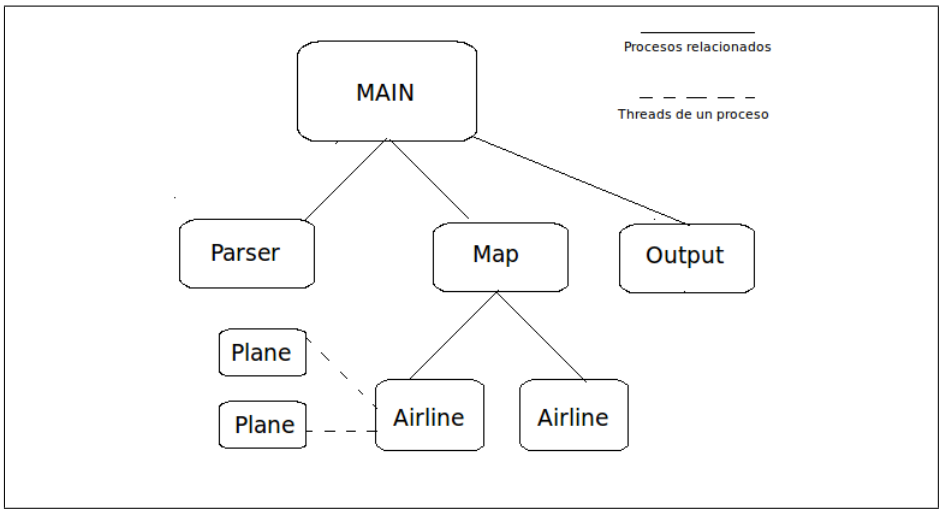

Figura 1: Esquema inicial de nuestra aplicación.

Luego de cierto debate, vimos que la relación entre el flujo principal, los parsers, el *output* y el mapa era demasiado fuerte, pues, la combinación de estas tres cosas iban a controlar el flujo de la simulación en sí. Inclusive, estas partes tenian una secuencia bastante marcada. Así, se decidió que estos elementos que originalmente bien podrían haber sido planteado como procesos paralelos, era más intuitivo y coherente escribirlo como uno solo. Finalmente, el mapa y el *output* fueron implementados como threads del proceso principal ya que si bien su relación es fuerte, nos resulto coherente que ambos puedan estar ejecutandose paralelamente.

Por otra parte, la aerolínea y los aviones nos resulto intuitivo implementarlo como un proceso aparte. La siguiente decisión tomada fue que los aviones debian ser *threads* de los procesos aerolinea. Pues nos resulto necesario implementarlo de esta manera ya que la relación entre una aerolínea y sus aviones era constante, ya que un avión es parte de una aerolínea. Aquí se presente otro

esquema del modelo final

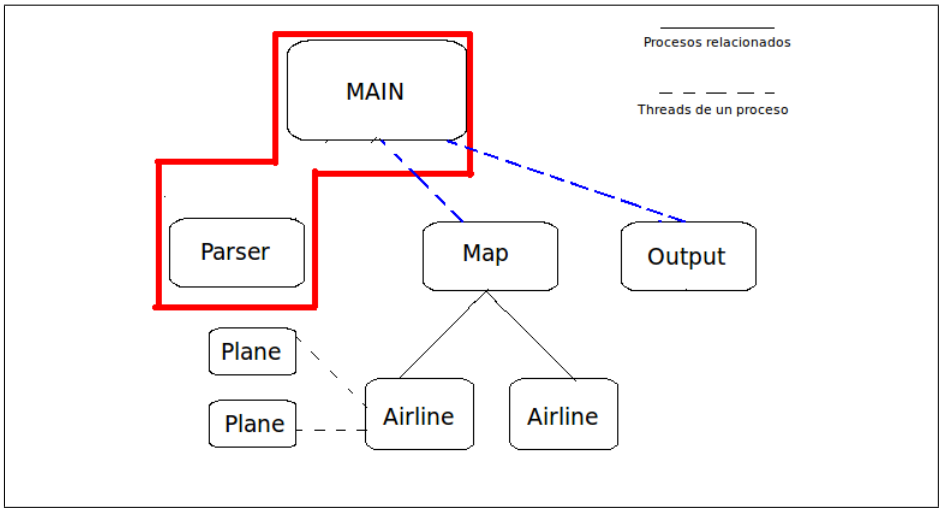

Figura 2: Esquema final de nuestra aplicación.

3. Modelo OSI

El modelo OSI es un modelo que estandariza la forma de diseñar un programa y como comunicar diferentes programas. El modelo OSI cuenta con 7 capaz aunque dadas las funcionalidades de nuestra aplicación solamente debimos implementar cuatro (e inclusive, dos de las mismas están implementadas como una). Esto se debe a que nuestro programa no se comunica con otros programas, reduciendo así las capas del modelo OSI implementadas por nosotros. El resto de las capas vienen implementadas por el sistema operativo.

El modelo OSI es muy útil ya que permite realizar las diferentes partes de una aplicación por separado sin mezclar información innecesaria y de esta manera las implementaciones en cada capa son independientes del resto. Por último, la separación en capaz mejora ampliamente la claridad del código.

En nuestro caso, la capa de aplicación es la encargada de realizar el flujo y la lógica de la aplicación en sí. Las capas de presentación y sesión estan implementadas como una sola, la capa de Marshalling. Esta empaqueta los datos a transmitir a otro proceso en formatos determinados. Nuestra última capa implementada es la de transporte, que es la que se encarga de utilizar los diferentes tipos de *IPCs* para comunicar procesos.

Notesé que es de suma importancia la separación en capas ya que, por ejemplo, gracias a esto fue posible implementar los distintos tipos de *IPCs* sin importar el resto de las implementaciones.

En el esquema presentado a continuación se puede observar lo descripto en esta sección.

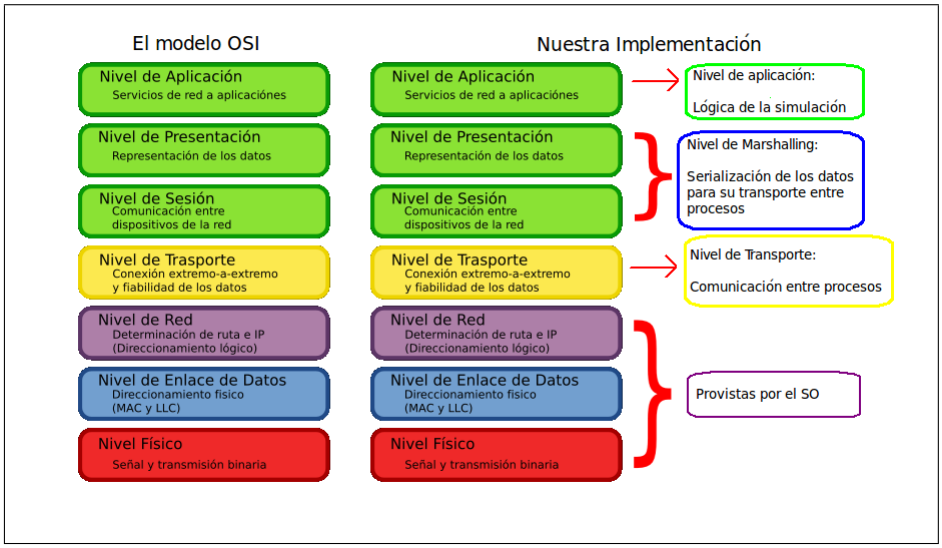

Figura 3: Esquema del modelo OSI y nuestra implementación del mismo.

4. Interfaz de comunicación(???)

Presentar las distintas interfaces que tuvimos... Creo que fueron tres en total... Criterios de elección Y de descarte... CHAMPO esta es TU parte, te llama a gritosss jeje.

5. Problemas encontrados(???)

Uno de los primeros problemas con los que nos topamos, fue el hecho de que los archivos de configuración de los cuales la aplicación debía levantar la información, tenían un formato bastante díficil para trabajar con el lenguaje C. Al no contar con una estructura que pueda ir agregando elementos a una colección y cambiando de tamaño dinámicamente, debimos implementar nuestra propia estructura, *Vector*. Esto probó ser muy útil más adelante ya que la utilizamos en otros sectores del programa.

Otro problema con el formato de los archivos de configuración, fue que los límites de las "iteraciones" se marcaban con líneas en blanco, y al parecer hay problemas al querer leer el $\backslash n$ con $fscanf$. Para lidiar con esta situación, recurrimos a la única solución que encontramos, aunque en términos de código no es muy elegante.

Otro problema encontrado, se relaciona con el testeo. Una vez implementadas las 4 implementaciones de la interfaz de IPC, diseñamos un pequeño test inicial, para detectar fallas en una etapa temprana. El test no era del todo básico, pero tampoco era muy agresivo. En ese momento, nos concentramos en que las implementaciones efectivamente funcionaran (comunicaran procesos). Al no haber hecho un test más exhaustivo, a la hora de testear la aplicación completa como un todo, surguieron algunos pequeños problemas relacionados con la comunicación entre procesos que nuestro test inicial no había logrado detectar.

6. Conclusiones

Queda claro que este trabajo podría haberse realizado sin necesidad de recurrir al procesamiento paralelo y a la comunicación entre procesos. Pero al haberlo hecho de de esta forma, la experiencia adquirida y el aprendizaje resulta mucho más significativo. Por varias razones:

- la experiencia adquirida al trabajar con varios procesos corriendo al mismo tiempo, al igual que varios *threads*.
- la comunicación entre estos procesos.
- la separación en capas, consecuencia necesaria, que agrega mucha claridad al código y conceptualmente, al diferenciar claramente las responsabilidades.
- la familiarización con los estándares *POSIX* y *System V*, o el simple de hecho de trabajar contra interfaces predefinidas y respetando un estándar predefinido.

Con respecto al uso de *threads* y procesos, nos topamos con una fuerte diferencia entre utilizar unos y otros. Por ejemplo, el uso de variables globales en un *thread* debe ser llevado a cabo con mucho cuidado, ya que el estado global, o *Data Segment*, es compartido por los demás *threads* del mismo proceso. Esto no es así con los procesos, cuyo estado global es propio y solo visible a ellos mismos. Por suerte, el *stack* no es compartido por distintos *threads* los que les da un cierto grado de independencia, para que realicen distinas tareas. Otra diferencia que notamos, tiene que ver con la sincronizacíon. Mientras que un simple *lock* de un *mutex* en la mayoría de los casos era lo único que se debía hacer para sincronizar *threads*, para procesos se requirió el uso de técnicas más avanzadas como semáforos.

Una vez terminado el trabajo, surgió la pregunta de cual era la implementación de IPC más rápida. Como nos dimos cuenta más tarde, esto no es tan fácil de determinar. Esto se debe a que el rendimiento de cada implementación varía dependiendo de la ejecución debido al procesamiento paralelo y el cual no es determínistico. Es decir, no es siempre se reproduce exactamente la misma ejecución dadas las mismas condiciones iniciales.

A pesar de lo dicho anteriormente, es posible, observar ciertas tendencias en los tiempos de ejecución. El siguiente gráfico muestra los distintos tiempos obtenidos por cada implementación para una serie de 4 simulaciones. La simulación 1 es la que presenta los menores tiempos, con 5 ciudades, 2 aerolíneas y 8 aviones en total. La simulación 2, presenta una configuración similar aunque, en general, tarda más en terminar debido a una menor cantidad de aviones. Sus parámetros son 5 ciudades, 1 aerolínea y 3 aviones. La simulación 3, cuenta con 10 ciudades, 10 aerolíneas y 40 aviones en total, por lo que el tiempo de ejecucón, como es de esperar es un poco superior. Finalmente, la simulación 4 es la más intensiva, ya que tiene la configuración límite, con 50 ciudades, 10 aerolíneas, y un total de 100 aviones. Esta es, como se puede ver, la que presenta los mayores tiempos de ejecución.

Cabe aclarar, que debido a la variación que sufren de ejecución en ejecución nos pareció prudente tomar el promedio de 100 ejecuciones para cada simulación.

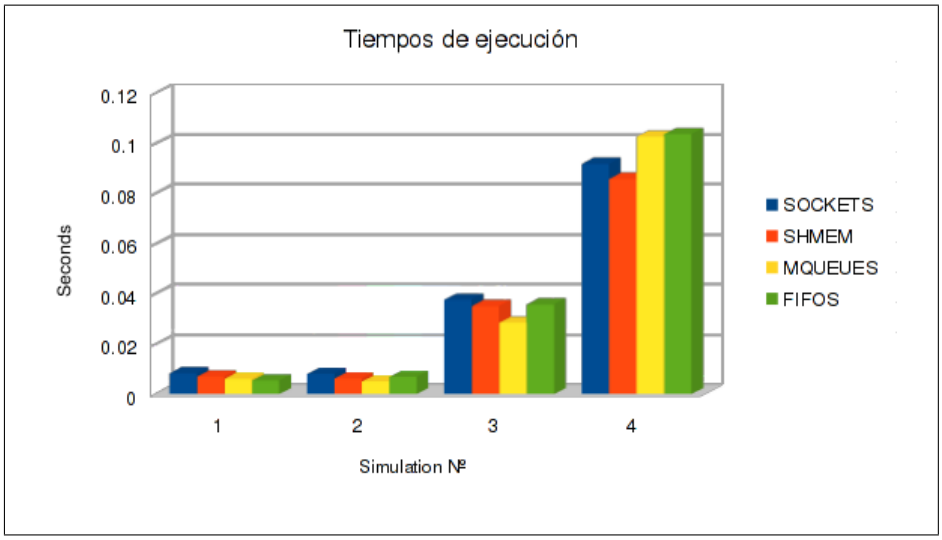

Figura 4: Comparación de los tiempos de ejecución de cada implementación.

En base a estos datos, pudimos concluir que en general la de *shared memory* es la implementación con mejores tiempos, seguido de la de *socket*. Aunque, como puede observarse, para las primeras simulaciones, las menos intensivas, las implementaciones de *message queues* y *fifos* parecen tener mejores tiempos. Después de analizar los datos, nuestra explicación es que, en esos casos el mayor *overhead* que tienen las implementaciones como *socket* y *shared memory*, hacen que sus tiempos de ejecución sean levemente mayores. Pero, para simulaciones más intensivas, como por ejemplo la número 4, ese *overhead* se torna despreciable, revelando los resultados que se observan en el gráfico: esas implementaciones terminan siendo más rápidas.

7. Referencias

- Material provisto por la cátedra.
- UNIX system programming. Second Edition. Keith Havilland, Dina Gray, Ben Salama.
- http://cplusplus.com/cir
- http://beej.us/guide/bgipc/output/html/multipage/unixsock.html
- https://computing.llnl.gov/tutorials/pthreads/
- https://computing.llnl.gov/tutorials/parallel_comp/
- http://www.csc.villanova.edu/ mdamian/threads/posixsem.html
- http://www.cs.cf.ac.uk/Dave/C/node25.html
- http://www.users.pjwstk.edu.pl/ jms/qnx/help/watcom/clibref/mq_overview.html
- http://linux.die.net/man/7/mq_overview