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

Esquema onda el que hicimos con bruno sobre que cosas son procesos y que son threads, y que cosa se comunica con que. Aunque sea un paint.

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, a las aerolíneas y a los aviones. Luego de cierto debate, vimos que la relación entre el flujo principal, los parsers 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.

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 que 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.

A continuación se presentan esquemas que representan las tomas de decisiones respecto al esquema de la simulación recién mencionadas:

3. Modelo OSI

Separación en capas. Separación clara de responsabilidades. Mejora en la claridad del código. Esquema gráfico.

El modelo OSI es un modelo que estandariza la forma de diseñar un programa y como comunicar diferentes programas. Como se puede ver en el esquema (INCLUIR ESQUEMA)
el modelo OSI cuenta con 7 capaz aunque dadas las funcionalidades de nuestra aplicación nuestro programa solo cuenta con las primeras cuatro implementadas (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 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 se encarga de recibir los datos que se desean transmitir a otro proceso y dejarlos en estructuras de modo tal que la capa de Marshalling de otro proceso lo pueda entender. 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.

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 ïteraciones"se marcaban con líneas en blanco, y al parecer hay problemas al querer leer el
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

ACUERDENSE DE COMO NOS LA HIZO COMER ROMANNNNN!!!PONGAMOS ALGO COPADO ACA!!!XD

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

Estaria bueno poder comparar el programa usando los diferentes tipos de ipc

7. Referencias

- Material provisto por la cátedra.
- UNIX system programming. Second Edition. Keith Havilland, Dina Gray, Ben Salama.
- http://cplusplus.com/
- 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