Sistemas Operativos

ITBA
Cuatrimestre 1 - 2012

Filesystems, IPC's y Servidores Concurrentes

Autores:

- Mader Blanco, Conrado — Legajo: 51270
- Ramundo, Federico — Legajo: 51596
- Mehdi, Tomás — Legajo: 51014

1 - Introducción

A partir de las clases prácticas y teóricas se llevara a cabo un sistema cliente-servidor de una liga fantasía.

2 - Objetivo

El objetivo de este trabajo es familiarizarse con el uso de sistemas cliente-servidor concurrentes, utilizando distintos hilos de ejecución y procesos que se comunican entre sí mediante cuatro tipos de IPC (*Inter Process Comunication*) distintos, y manejar con autoridad el filesystem de Linux.

Esto se llevó a cabo mediante la creación de procesos hijos (fork) y mediante la creación de threads. Los cuatro tipos de IPC utilizados fueron FIFOs del filesystem, message queues, shared memory y sockets UDP.

3 - Diseño e Implementación

3.1 - Estructuras

Se crearon las siguientes estructuras de datos:

- user_t contiene:
  + ID
  + lista de equipos
  + nombre
  + contraseña
  + ID de la liga que esta en draft en ese momento

- client_t contiene:
  + ID
  + el usuario al cual corresponde
  + el thread que se ejecuto para atender a este cliente
  + el thread por el cual se comunica para indicar que sigue conectado
  +el tiempo que lleva sin mandar un mensaje (para control de conexión)
  + un canal de comunicación (IPC)

- team_t contiene:
  + ID
  + nombre
  + un puntero a usuario
  + un puntero a liga
  + puntaje

- sportist_t contiene:
  + ID
  + nombre
  + puntaje
  + el equipo al que pertenece

- draft_t contiene:
  + la liga a la que pertenece
  + un vector de punteros a cliente
  + un vector de semáforos
  + un flag
  + el turno de a quien le toca elegir
  + más información acerca del tiempo de cada turno y si se envió o no la información al jugador que le toca elegir

- league_t contiene:
  + ID
  + ID del siguiente equipo que debe unirse
  + nombre de la liga
  + contraseña
  + un vector de deportistas
  + un vector de punteros a equipos
  + la cantidad de equipos actual
  + la cantidad máxima de equipos
  + una lista de intercambios
  + ID del siguiente intercambio que se generará
  + un draft (vale NULL si ya se realizó)

- trade_t contiene:
  + ID
  + la liga a la que pertenece
  + el equipo que quiere realizar el intercambio
  + el equipo que recibe la oferta

+ el deportista ofrecido
+ el deportista por el cual se pide el cambio

En algunos casos guardar referencias circulares (como sucede con usuario y equipo) se consideró oportuno a la hora de manejar los datos.

Se decidió que sea el usuario el que tenga el ID de la liga que esta en draft en ese momento dado que resulta más fácil, si se desconectó en el proceso, controlarlo en el usuario en el momento que hace el *login* más que iterar por sus equipos a ver si alguno está en un draft. Cuando no se encuentra en un draft esta variable vale -1.

En la estructura draft se decidió guardar mucha información acerca de su estado (tiempo, turno, etc...) dado que en muchas ocasiones es necesario acceder a esta información (si un usuario se desconecta y desea reconectarse) y resulta más cómodo acceder directamente a las variables que tener que preguntar por ellas. Dado que el draft existe sólo en la etapa más temprana de la liga no se consideró que fuese un desperdicio de espacio porque su tiempo de vida es corto.

3.2 - IPC

Para poder lograr que tanto el cliente y el servidor desconozcan qué IPC se está utilizando fue necesario crear un patrón de funciones, la cuales se implementan de manera diferente en cada IPC. Para poder llevar esto a cabo manejamos los canales de comunicación como un puntero a void abstracto. En cada caso guarda una estructura formada por:

- FIFOs: dos enteros (cada uno es un file descriptor) correspondientes a el canal de lectura y el de escritura.

- Message Queues: un entero que hace referencia a la cola, y dos más que corresponden al ID de escritura y al de lectura.

- Shared Memory: se guarda el puntero a la zona de memoria compartida y dos semáforos para controlar la zona de escritura y lectura.

- Sockets: un entero correspondiente al socket propio, otro que es el ID de nuestro canal y por último una dirección de donde leer y escribir.

Las funciones que implementan cada uno de los cuatro IPC son las siguientes:

- createChannel

- connectChannel
- sndMsg
- rcvMsg
- sndString
- rcvString
- disconnect
- destroyChannel

3.3 - Funcionalidad

3.3.1 - Servidor

El servidor cuenta con varios hilos de ejecución en paralelo (threads) que realizan las siguientes funciones:

- **Escuchar a un nuevo cliente:** Este hilo de ejecución constantemente lee el canal default a la espera de nuevos clientes. En cuanto encuentra uno nuevo, le asigna un canal de comunicación y le delega la atención a otro hilo de ejecución. Además este thread es el encargado de controlar que los clientes no se hayan desconectado, por medio de este canal default.

- **Atender a un cliente:** Se encarga de realizar el login del cliente y proporcionarle la información necesaria al cliente para realizar las diferentes operaciones. Cuando el cliente se desconecta se encarga de destruir el canal de comunicación entre estos.

- **Draft:** Realiza todo lo pertinente a la elección de deportistas para los equipos de nuevas ligas. Este hilo de ejecución se explica con detalle más adelante.

- **Imprimir en pantalla:** Dado que se poseen tantos hilos de ejecución en paralelo es necesario un thread especifico para escribir la información relevante del log en pantalla de forma ordenada. Esto se realiza mediante una cola de mensajes, en la cual cada thread, cuando desea escribir, inserta una cadena de caracteres y levanta un semáforo, para que este thread de impresión lo desencole y lo imprima.

- **Cargar partidos nuevos:** Este thread controla un directorio en donde se cargan los partidos de cada liga. Cuando el thread encuentra un archivo lo lee, realiza la asignación de puntos pertinente, y elimina el archivo.

- **Guardado de estado:** Cada cierto tiempo, es necesario guardar el estado de las ligas. Este thread realiza este trabajo, teniendo cuidado, mediante la implementación de semáforos, de no leer alguna variable que esta siendo modificada en ese momento.

El servidor posee variables globales que todos sus hilos pueden ver y escribir. Estas variables son:

- Array de punteros a ligas
- Array de punteros a usuarios
- Lista de clientes conectados
- La cantidad de ligas y de usuarios
- Los ID a asignar a las siguientes ligas, usuarios y clientes creados
- La cola de mensajes para la impresión

La mayoría de estas variables se inicializan cuando se ejecuta el servidor, mediante la función que carga la información de los archivos guardados.

3.3.2 - Cliente

Tras haber conectado con el servidor y que este le asigne un canal de comunicación el cliente entra en un diálogo con su hilo de ejecución del servidor asignado. Cada vez que realiza un comando, la shell del cliente ejecuta un proceso hijo correspondiente a dicho comando (mediante la función *fork*). Mientras el proceso hijo se ejecuta, el padre espera a que termine (mediante la función *wait*), emulando así el comportamiento de una shell tradicional. Cada proceso hijo, correspondiente a cada uno de los comandos soportados, se encarga de dialogar con el servidor, para que este le envíe la información. Al comenzar, el cliente le manda un entero, correspondiente al tipo de información que quiere, que el servidor descifra según un protocolo definidos como constantes en common.h para poder atender a su cliente.

En el caso del proceso del draft, se posee, al igual que en el servidor, un thread aparte que escucha el deportista escogido, mientras el hilo original controla que no se haya excedido de tiempo.

El cliente además posee un hilo de ejecución aparte que se encarga de comunicarle al servidor cada cierto tiempo que esta “vivo”, es decir, que no se desconectó.

Existe un factor de conversión para los ID de los deportistas, las transacciones y los equipos del lado del cliente. El cliente ve que cada una de estas estructuras tiene su ID más el ID de la liga a la que pertenece multiplicado por mil. Por ejemplo, el deportista ID 3 de la liga con ID 5 será el deportista con ID 5003 para el usuario. Se tomó esta decisión dado que cuando un usuario desea indicar con qué deportista,

transacción o equipo interactuar, debe indicar su ID, lo que puede llevar a confusión si existen más de una liga (existirían muchos equipos con ID 1 para el usuario, por ejemplo). Otra forma de hacerlo habría sido preguntar después de indicar el ID de dicha estructura, a qué liga nos referimos pero decidimos realizarlo de la otra manera dado que nos ahorramos el pasaje de un mensaje cada vez.

Los comandos soportados por el cliente son:

- **Sign up [user, password]**

Crea un usuario nuevo

- **Log in [user, password]**

Entra al sistema con un usuario

- **Create League [name, password]**

Crea una liga. Puede tener contraseña o no.

- **Join League [league ID, password]**

Entra a una liga, siempre y cuando no haya empezado

- **Draft [league ID]**

Entra al draft en una liga

- **List [leagues, teams o trades]**

Da información acerca de todas las ligas, todos tus equipos y todos los intercambios en los que estas involucrado

- **League show [leagueID]**

Proporciona información acerca de una liga en particular

- **Team show [teamID]**

Proporciona información acerca de un equipo en particular

- **Trade show [trade ID]**

Proporciona información acerca de un intercambio en particular

- **Trade [team ID, offer, change]**

Genera un intercambio, siempre que poseas un equipo en la liga a la cual pertenece el equipo al que le deseas cambiar, que tengas en ese equipo a tu oferta y que él tenga en el suyo a tu interesado. Si se desea cambiar un deportista tuyo por otro que no tiene equipo se debe especificar con -1 en el *team ID.* El servidor es capaz de detectar a qué liga pertenece ese deportista dado al factor de conversión que comentamos anteriormente.

- **Trade accept [trade ID]**

Acepta un intercambio ofrecido por otro equipo.

- **Trade withdraw [trade ID]**

Retira un intercambio ofrecido por ti

- **Trade Negociate [trade ID, newOffer, newChange]**

Negocia un intercambio: cancela el existente y reformula otro siempre y cuando se cumplan las condiciones de generación de intercambios explicadas arriba.

3.4 - Funcionamiento del draft

Cuando una liga es creada, se le asigna un puntero a la estructura *draft* tal y como se explicó anteriormente. Esta estructura posee un vector de punteros a cliente, inicialmente lleno de NULL, y otro vector con semáforos inicializados con el valor 0.

Cuando un usuario ejecuta el comando de drafting a una liga a la que pertenece, se cambia un valor en el vector de clientes del draft, que pasa de valer NULL a el valor correspondiente a la dirección de dicho cliente. En este momento, el thread del servidor que atiende al cliente baja el valor del semáforo correspondiente a ese cliente en el vector del draft para bloquearse, hasta que se termine el draft, o hasta que el cliente desee salir del draft. Este usuario queda en estado de espera hasta que el draft comience, cuando todos los integrantes de la liga decidan realizar el draft. Es posible salir de la espera en todo momento.

En el momento que todos los equipos están dispuestos a comenzar el drafting, se crea un hilo de ejecución paralelo que atiende a los clientes, comunicándoles si deben esperar, o si es su turno de escoger un deportista, en cuyo caso les informa el estado de la liga en el momento. En cada turno, el proceso del servidor debe escuchar el deportista escogido por el usuario y a la vez calcular el tiempo que le queda a este para la elección. A su vez, del lado del cliente, se debe leer la entrada del usuario, y además controlar el tiempo, al igual que el servidor. El control del tiempo y la lectura/ escritura del deportista se realiza con dos hilos de ejecución paralelos, en ambos lados cliente-servidor.

Si el tiempo se agota sin que haya habido elección se asigna un deportista arbitrario.

Es posible salir del draft y reincorporarse en todo momento. Si el jugador está ausente en su turno, el hilo de ejecución que escucha la elección de este usuario se bloquea en un semáforo, hasta que este se reconecte, al draft, dándole una señal a dicho semáforo para despertar el thread que lo atiende. Cuando esto sucede, el cliente debe obtener del servidor la información de cuanto tiempo le queda disponible para realizar su elección en ese turno.

3.5 - Actualización de puntos

Para actualizar los puntos de los deportistas y los equipos contamos con un directorio en el cual se guardan archivos con la información de estos partidos. El hilo de ejecución del servidor correspondiente a este apartado, lee de este directorio hasta encontrar un archivo. Cuando esto sucede, asigna los puntos pertinentes a cada entidad y borra el archivo del partido.

3.6 - Persistencia

La información del estado del sistema se encuentra guardada en archivos. Lo primero que realiza en servidor al ejecutarse es cargar la información de dichos archivos, y durante la vida del proceso se guarda en ellos cada cierto tiempo.

Existe un archivo diferente para guardar información acerca de las ligas existentes, los usuarios creados, y los deportistas. A su vez, cada liga tiene archivos que dan información acerca de sus equipos, sus intercambios y sus deportistas (el estado que tienen en esa liga en particular). Estos archivos se identifican dado que en su nombre poseen el ID de la liga a la que corresponden.

Existen controles para evitar que se guarden variables que están siendo cambiadas en el momento (como cuando se realiza un draft).

4 - Problemas Encontrados

4.1 - Estructuras

En primer lugar, en las estructuras, cada equipo tenia a sus deportistas, y la liga sólo tenia los deportistas que no tenían equipo. Esto se complicaba a la hora de hacer intercambios por lo que se decidió hacer que la liga conozca a todos los deportistas, y que cada uno de ellos tenga su equipo, y no al revés.

4.2 - Canales de comunicación

Al comienzo del desarrollo se manejaron los canales de comunicación con un entero. En el caso de los FIFOs este entero era el File Descriptor. En el caso de las colas, era el Message Queue Descriptor devuelto al crearla. Esto funcionaba genéricamente para ambos IPCs, pero al momento de implementar Shared Memory ese entero simplemente no era suficiente, se necesitaba manejar semáforos y punteros a memoria. La solución encontrada fue utilizar un puntero a void como “canal” en lugar de un entero. Y cada implementación maneja internamente ese puntero como la estructura necesaria, funcionando como una caja negra (el servidor y el cliente simplemente crean un canal sin saber que estructura se utiliza internamente) y de

esta forma poder encajar los cuatro tipos de comunicación en un solo prototipo de funciones.

4.2.1 - Multiplicidad de canales

El primer acercamiento que se utilizó fue erróneo: utilizar dos colas de mensajes unidireccionales para cada conexión (con lo cual habría más de 10 colas de mensajes si se conectan 5 clientes). Lo mismo en el caso de Shared Memory (se asignaban dos zonas por cada cliente).

La solución en el caso de Message Queues fue utilizar una única cola, que maneje mensajes con prioridades para identificar con quien nos estamos comunicando.

En el caso de Shared Memory se reservó una única zona de memoria compartida, subdividida en distintos buffers. Cada vez que un cliente se conecta se le asignan dos buffers de esta zona de memoria compartida: uno para escritura y uno para lectura.

4.2.2 - Message Queues: POSIX vs SystemV

Al solucionar el problema de la multiplicidad de canales, se encontró una limitación en las colas de mensajes de POSIX: no se pueden recibir mensajes con una prioridad especifica, sino que solo pueden pedirse los mensajes con una prioridad o superior. Fue necesario cambiar la implementación y utilizar las Message Queues que provee SystemV.

Surgió otro problema con las colas, a la hora de mandar un double (en el draft, para indicar cuanto tiempo queda de turno) que se mandaba un cero, y hacia creer que el código esperaba en un scanf, dando muchos dolores de cabeza.

4.2.3 - ¿Un canal para escribir y otro para leer?

No fue hasta implementar Sockets que hubo que preguntarse qué era lo que realmente convenía. En un primer momento se utilizaban dos canales, uno para escribir y otro para leer, no importa que IPC se utilizara. Esto exponía comportamiento de los IPC's a una capa superior. Probablemente se hizo de esa manera porque el primer IPC implementado fue FIFO, que realmente es unidireccional y necesita dos pipes para una comunicación bidireccional. La solución fue implementarlo como un solo canal en la capa superior; y que la capa de marshalling sea la que se ocupe de almacenar dos canales separados si es necesario. Esto permitió utilizar un código mucho más claro en el servidor y el cliente, abstrayéndose del IPC utilizado.

4.2.4 - Shared Memory: bloqueos

Shared Memory fue sin dudas el IPC en el que hubo que implementar más funcionalidades “a mano”. Se resolvió utilizando buffers circulares para enviar mensajes en cada dirección. Al momento de leer del buffer, primero se utilizó un ciclo que esperara que realmente haya algo escrito para leer (es decir, que la cabeza del buffer se adelante a la cola). Esto funcionaba, pero se ignoraba que esta misma funcionalidad de “bloquearse” era mucho más eficiente utilizando un semáforo adicional al ya existente (de exclusión mutua). Se implementó este nuevo mutex reduciendo los posibles problemas a futuro y también reduciendo el código en la lectura del buffer.

Más adelante, nos dimos cuenta que también necesitábamos otro mutex para que el escritor que bloquee si el buffer estaba lleno, tal y como en el ejemplo visto en clase del productor-consumidor.

4.2.5 - Sockets

4.2.5.1 - TCP vs UDP

La primera implementación de sockets se realizó utilizando sockets TCP. Se tomó esta decisión dado que se creía conveniente para n sistemas cliente-servidor, pero surgieron problemas para integrarlo con las funcionalidades ya implementadas por los otros 3 IPC. Por este motivo, de decidió abandonar los sockets TCP para decantarnos por los UDP, dado que encaja con nuestro formato de comunicación, y no se perdían otras ventajas del TCP en nuestro caso, como la seguridad de que lleguen los mensajes a destino.

4.2.5.2 - Unix vs Inet

En primer lugar utilizamos direcciones UNIX. Se lograron enviar mensajes pero se encontraron problemas a la hora de almacenar la dirección del emisor de un mensaje. La solución más simple fue utilizar direcciones de internet, las cuales parecen ser soportadas por todos los sistemas (no así las de UNIX).

Otro problema que surgió con este IPC surgió por el binding. En primer lugar realizábamos binding tanto desde el servidor como desde el cliente. Además de estar mal conceptualmente, nos encontrábamos con un problema cada vez que en el cliente se generaba un proceso hijo, dado que debía hace el *unbind* el padre para que este se pueda comunicar. Finalmente lo resolvimos haciendo que solo realice el bind el servidor, y que este escuche el primer mensaje del cliente, guardándose su dirección con la función *recvfrom.*

4.3 - Ejecución y parsing de comandos

4.3.1 - Ejecutar comandos en el servidor: switch-case

A medida que se implementaron más y más comandos, el código en el servidor que se encargaba de recibir el mensaje del cliente y ejecutar el comando correspondiente se convirtió en un *switch-case* enorme. La solución fue implementar un vector de punteros a función y separar el código de ese switch en distintas funciones (una por cada comando). El código del servidor se redujo a simplemente llamar a la función de ese vector en la posición correspondiente.

4.3.2 - Parser de comandos en el cliente

Al igual que en el servidor, al implementar más comandos el código en el cliente se incrementó debido a que era necesario realizar un *parsing* de la entrada estándar para buscar coincidencias con los comandos implementados. Este código se separo a un fichero *shell.c* que se encarga simplemente de leer la entrada estándar y ejecutar el programa correspondiente. No se encontró una solución tan efectiva como el vector de punteros a funciones porque en algún punto se debe realizar el trabajo sucio de comparar strings por cada comando.

En otros lenguajes se podría haber utilizado una implementación de Tries, pero en este caso no valía la pena puesto que deberíamos implementarlo nosotros o adaptar alguna implementación de un libro/internet.

4.4 - Draft

En general podemos decir que la parte correspondiente al draft fue la más problemática de todo el trabajo. Fue costoso por el hecho de que debíamos estar comunicándonos con muchos clientes a la vez, controlando que no se hayan desconectado, integrándolos nuevamente.

Nos costó darnos cuenta que en primer lugar el draft no funcionaba, porque había otro proceso “consumiéndole” los mensajes que el cliente le mandaba al draft, dado que la herramienta gdb no es muy útil para programas multithread y multiproceso.

Nos ocasionó problemas el hecho de que lo habíamos diseñado de una manera, y a último momento debimos cambiarla: En primer lugar era imposible salir del draft, salvo que el proceso del cliente se muera. Si éste se reconectaba ocasionalmente, era obligado a regresar al draft. Finalmente debimos agregar la posibilidad de salir del draft.

Otro problema que nos encontramos en este caso fue que, los días antes de la entrega nos dimos cuenta que estábamos realizando lo llamado *busy waiting* por medio

de un ciclo *while* que no terminaba hasta que cambiaba una variable, en los casos que debíamos esperar (como cuando el thread del servidor que atiende al cliente debía bloquearse hasta que termine el draft). Por suerte, se nos ocurrió utilizar semáforos para esta espera, que bloquean el proceso, haciendo que no consuman CPU.

4.5 - Debugging

Nos encontramos con problemas en general a la hora de realizar el *debugging* dado que la herramienta gdb no funciona tan bien en programas multiproceso y multithread. Si bien se puede realizar, es muy incómodo y resultaba más práctico buscar errores del código mediante impresiones a pantalla utilizando printf.

4.6 - Curses

Intentamos realizar del lado del cliente una interfaz con la librería curses, pero una vez completada nos resultó incómodo el hecho de que no era posible realizar un *scroll* hacia arriba si el servidor proporcionaba mucha información junta (como en el caso del draft) por lo que terminamos por desecharla. Esto fue un problema dado que resultó una perdida de tiempo.

5 - Conclusión

A pesar de las adversidades y falta de conocimiento al inicio de la materia se logro el objetivo del trabajo práctico, que en definitiva era familiarizarnos con IPC's y el formato cliente-servidor para poder lograr los siguientes Trabajos Prácticos Especiales que por definición son más difíciles. Puede haber resultado laborioso a su vez el hecho de no ser tan expertos en el lenguaje c y el manejo de punteros, al estar más acostumbrados a programar con Java.

6 - Referencias

- Brian W. Kernighan, Dennis M. Ritchie, *El lenguaje de programación C*

- Kurt Wall, *Programación en Linux con ejemplos*
- Keith Haviland, Dina Gray, Ben Salama, *Unix System Programming (2nd Edition)*
- Etchegoyen, Hugo Eduardo, *Archivo word*