|  |  |
| --- | --- |
|  | FantasySO - The Game |
|  | [v1.0]  *Alan Karpovsky - María Victoria Mesa Alcorta - Facundo Nahuel Martinez Correa* |

|  |
| --- |
| **[**Filesystems, IPCs y Servidores Concurrentes**]** |
| El presente documento tiene por objetivo analizar y detallar la implementación de una liga de fantasia en el lenguaje C hacienda uso de distintos tipos de comunicación entre procesos como así también de la noción de servidores concurrentes. |

**Sistemas Operativos**

**Abril de 2012**

# Índice

Contenidos

Índice 0

1. Introducción 1

Objetivo 1

2. Estructura de archivos 2

Modelado de la base de datos 2

User 3

League 3

3. Threads y procesos 4

Threads 4

Atención de nuevos clientes 4

Coordinación del DRAFT 5

Monitoreo de partidos 5

Persistencia 6

Procesos 6

4. IPCs 7

Generalidades 7

Estructuras de mensaje 8

Mensaje servidor --> cliente *(msg\_s)* 8

Mensaje cliente --> servidor *(msg\_t)* 9

FIFOs 11

Message queues 12

Sockets 13

Shared Memory 15

5. Serialización de datos 18

6. Draft 19

7. Conclusiones 20

# Introducción

El presente trabajo consiste en la implementación de una **Liga de Fantasía** desarrollada en *Lenguaje C.*  Básicamente la aplicación es de tipo cliente/servidor en la cual los clientes son los usuarios que desean jugar a este juego y el servidor es el encargado de coordinar las peticiones de los mismos, manejar los archivos, persistir los datos, etc.

Desde el lado del cliente, al ejecutarse, la aplicación mostrará una shell similar a la de Linux con la cual el cliente deberá interactuar para ejecutar las acciones deseadas. Las mismas incluyen comandos para *loguearse* al sistema, consultar el estado actual de sus equipos, ver las ligas activas, unirse a una liga y otras operaciones que detallaremos en futuras secciones.

Por su parte el servidor, al ejecutarse, dejará el terreno preparado para que nuevas conexiones de clientes puedan establecerse. Tendrá como tarea principal responder a las peticiones de los clientes dándole los datos que éstos le solicitan y como tarea secundaria hacer el manejo del filesystem en el que se incluye la persistencia de datos como así también la carga de nuevos archivos de configuración de jugadores y partidos.

## Objetivo

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.

# Estructura de archivos

Como primer paso para el delineamiento del trabajo práctico, se buscó crear una estructura de directorios y archivos para modelar la base de datos. Se buscó una solución simple pero efectiva en la que existe una carpeta de resourses */res*. Que dentro de ella contiene una subcarpeta *users* que aloja a los archivos de usuario y otra subcarpeta *leagues* que dentro contiene subcarpetas en las cuales se guardan cada una de las ligas.

## Modelado de la base de datos

res/  
├───users/  
│   ├───Alan.user  
│   ├───Vicky.user  
│   └───Facu.user  
├───leagues/  
   └───1/ // *"Liga 1" (Tiene equipos Alan y Facu y 4 trades)*  
       ├───1.league  
       ├───teams/  
       │   └───Alan.team  
       │   └───Facu.team  
       ├───trades/  
       │   └───1.trade  
       │   └───2.trade  
       │   └───3.trade  
       │   └───4.trade  
   └───2/ // *"Liga 2" (Tiene sólo al equipo Vicky y 2 trades)*  
       ├───2.league  
       ├───teams/  
       │   └───Vicky.team  
       ├───trades/  
       │   └───1.trade  
       │   └───2.trade

Las **extensiones** de los archivos sirven a modo de guía para saber a qué refiere el contenido alojado en ellos. Por ejemplo, un archivo con extensión ***user*** contendrá dentro los datos de un usuario.

### User

(Ej. Alan.user)

Alan // Username  
123456 // Password  
3 // Número de ligas a las que pertenece  
1 // ID "Liga 1"  
23 // ID "Liga 23"  
115 // ID "Liga 115"

### League

(Ej.  Carpeta "leagues" tiene una subcarpeta por cada liga.)  
  
**Liga "1"**

*leagues/1/****1.league***

1 // LeagueID  
0 // League Status (ACTIVE / NOT ACTIVE)  
2 // Número de jugadores ("messi") libres (no drafteados)  
Messi // Jugador  
100 // Puntos Messi  
Xavi // Jugador  
10 // Puntos Xavi

*leagues/1/trades/****X.trade*** *(un archivo .trade por cada trade)*

X // TradeID  
2 // Trade Status (WAITING / ACCEPTED / REJECTED)  
Alan // UserID from  
Vicky // UserID to  
Messi // Jugador from  
Palermo // Jugador to

*leagues/1/team/****Alan.team***

2 // ID del team  
55 // Points  
3 // Número de jugadores  
Messi // Jugador  
23 // puntos de messi  
Palermo // Jugador  
10 // puntos de palermo  
Xavi // Jugador  
5 // puntos de xavi

# Threads y procesos

Uno de los requerimientos provistos por la cátedra para este trabajo en particular era que el mismo debía utilizar nociones tanto de threads *(POSiX)* como de procesos a través del *fork()*. Nuestra implementación hace uso de threads en el servidor para la atención de los clientes y para otras funciones y utiliza el *fork()* para .....

## Threads

### Atención de nuevos clientes

El diseño de nuestro servidor es bien sencillo: al ejecutarse el mismo corre en un único thread esperando nuevas conexiones de los clientes. Cuando un cliente se corre éste enviará un mensaje de tipo *CONTACT* al servidor y aguardará a que éste último le conteste. Asimismo el servidor, al recibir este mensaje, creará un nuevo thread exclusivo para atender la conexión con ese cliente en particular y seguirá escuchando en el otro thread por otros clientes que puedan llegar a desear conectarse.

A fin de mejorar la performance, el servidor principal encola cada una de las peticiones de conexión y las va atendiendo a medida que tiene tiempo.

El thread que atiende con exclusividad a un cliente, apenas es creado, establece un canal de comunicación único para ese cliente (según el IPC con el que se lo haya compilado), inicializa una varialbe con todos los datos del usuario (para dar respuesta rápida a las consultas de éste) y luego simplemente se encarga de escuchar comandos ***( Msg\_t IPCListen() )*** y luego ejecutarlos. Todos los comandos que llegan al servidor son comandos válidos ya que el cliente, antes de permitir al usuario comunicarse, valida que lo ingresado sea un comando válido y los parámetros sean del tipo correcto.

Llamamos **servidor principal** al servidor encargado de resolver las conexiones de nuevos clientes y **servidores secundarios** a los threads que atienden a cada uno de los clientes una vez establecida la conexión.

### Coordinación del DRAFT

Afasfasfas

### Monitoreo de partidos

Apenas se inicia el servidor principal, antes de comenzar a atender pedidos de nuevas conexiones, se crea un thread para monitorear la carpeta de partidos. Este thread básicamente lo que hace es, cada una determinada cantidad de tiempo establecida en un *define*, chequear la carpeta de partidos y ver si hay nuevos archivos. En caso de existir un nuevo partido el mismo es "jugado" y los puntajes de todos los usuarios pertinentes son actualizados.

Para lograr monitorear un directorio por cambios, hacemos uso de una función implementada en Linux llamada **inotify()** la cual genera un evento apareado con un directorio. Esto produce que, ante cualquier cambio existente en el directorio en cuestión, la función es notificada.

### Persistencia

Un punto no menos importante para esta aplicación es la persistencia de los datos. No tendría mucho sentido la implementación de una liga de fantasía sin considerar que los usuarios, las ligas, los partidos, los jugadores y demás elementos que componen a este juego sean salvados para luego ser cargados en un futuro y continuar con el juego.

Con el fin de cumplimentar esta tarea, el servidor principal crea un thread encargado de la persistencia de los datos. El funcionamiento de este thread es similar al del punto anterior: el thread, cada cierto tiempo, baja a disco todos los datos del juego contenidos en una variable "game" de tipo *Game\_t*.

Un punto para destacar es que las listas contenidas dentro de "game" (ligas, jugadores, equipos, usuarios, etc) deben ser bloqueadas a la hora de hacer el save ya que sino se podrían generar datos inconsistentes. Se utiliza un mutex que bloquea la variable y el thread, una vez finalizada la bajada a disco de los datos, desbloquea el mutex. Si en medio de la ejecución del save algún usuario intenta modificar algún parámetro (ej. crea una nueva liga), la llamada será bloqueada hasta que el mutex sea liberado y recién allí será ejecutada la acción en cuestión.

Es interesante mencionar que la estrategia utilizada a la hora de salvar es simplemente reescribir (pisar) toda la información vieja y crear los archivos nuevos. Esto se hizo a modo de simplificación para no deber andar recordando qué archivos se modificaron y cuales no; simplemente el thread genera todos los archivos de nuevo y los salva en el disco rígido.

## Procesos

Se utilizó procesos en ......

# IPCs

## Generalidades

El trabajo debía incluír la implementación de cuatro tipos distintos de IPCs: **FIFOs, sockets, memoria compartida y colas de mensajes.** Quedaba a elección de cada equipo utilizar las llamadas de POSIX o de System V.

A modo de simplificar la programación y buscando no repetir código, se utilizó la estrategia de implementar una *interfaz* en la capa de transporte. La **interfaz *transport\_c.h*** *y* ***transport\_s.h*** contienen todas las funciones que cada IPC debe implementar desde el lado del cliente y desde el lado del servidor respectivamente.

En líneas generales las funciones de transport sirven, desde el cliente, para conectarse al servidor *connectToServer()* y para enviar y recibir mensajes *send/RcvMessage()* de forma bloqueante y no bloqueante. Por su parte, el servidor, implementa funciones para crearse y dejar el terreno listo para recibir conexiones *uPlink()*, escuchar a los clientes *IPCListen()*, crear canales de comunicación exclusivos 1-a-1 con cada cliente *create/establishChannel()* y por último poder enviar y recibir mensajes hacia y desde los clientes.

* Interfaz de transporte del **cliente**

Msg\_s communicate(Msg\_t msg);

Msg\_s \_rcvmessage(**void**);

Msg\_s rcvmessage(**void**);

**int** sendmessage(Msg\_t msg);

**void** connectToServer(**void**);

**void** sigint();

* Interfaz de transporte del **servidor**

**void** uplink(**void**);

Msg\_t IPClisten(Channel ch);

**int** communicate(Channel ch, Msg\_s msg);

Msg\_s establishChannel(Channel ch);

**int** sendmessage(Channel ch, Msg\_s msg);

Channel createChannel(Msg\_t msg);

**void** closeMainServer(**void**);

**void** closeConnection(Channel ch);

Msg\_t rcvmessage(Channel ch);

## Estructuras de mensaje

Con el fin de brindar mayor claridad semántica a nuestro código, se optó por crear dos estructuras de mensaje para pasar datos entre el cliente y el servidor y viceversa.

### Mensaje servidor --> cliente *(msg\_s)*

Este tipo de mensajes es bien sencillo, el mismo guarda una lista simplemente encadenada de los mensajes que quiere enviarle al cliente y a su vez tiene dos enteros para marcar errores o tipos distintos de respuesta. Dado que nuestro cliente es un "cliente bobo" y sólo envía comandos al servidor y espera una respuesta de este para luego imprimirla por salida estándar, bastó con tener una lista de mensajes que el servidor le envía. Es decir, por ejemplo si el cliente desea listar las ligas, el servidor creará los strings de respuesta en una lista y se la pasará a través del IPC al cliente; luego el cliente sólo debe recorrer esa lista e imprimir todos los mensajes que allí se encuentren.

**typedef** **struct** msg\_s {

**int** responseType;

**int** status;

List msgList;

}msg\_s;

### Mensaje cliente --> servidor *(msg\_t)*

A diferencia del punto anterior, el mensaje del cliente hacia el servidor es una estructura más compleja. En ella se declaran variables que contemplan todos los distintos tipos de comandos que el cliente podría llegar a ejecutar.

**typedef** **struct** msg\_t{

**int** type;

**union** data{

**char** **\*** tempnam;

**int** ping;

**int** ID;

**char** **\*** name;

**struct** register\_t{

**char** **\*** user;

**char** **\*** pass;

}register\_t;

**struct** login\_t{

**char** **\*** user;

**char** **\*** pass;

}login\_t;

**struct** show\_t{

**int** ID; */\* league, team o trade \*/*

}show\_t;

**struct** trade\_t{

**char** **\*** from;

**char** **\*** to;

**int** teamID;

**int** tradeID;

}trade\_t;

**struct** socket\_client\_t{

**int** client\_pid;

**struct** sockaddr\_in **\*** client;

}socket\_client\_t;

}data;

**int** pidFrom;

} msg\_t;

Nótese la utilización de **uniones** dentro de la estructura de mensaje. Las mismas nos permiten no desperdiciar espacio a la hora de alocar memoria. A su vez brindan mayor claridad al desrreferenciarla.

Nótese que varias de las funciones de la API de *transport* recibe o utiliza un **canal**.Lo que se hizo fue englobar todos los datos de un determinado cliente en una estructura **channel\_t**. De esta manera, las funciones del servidor como por ejemplo *IPCListen()* reciben un canal y se encargan de hacer lo que su nombre indica, escuchar, pero únicamente en el canal pasado como parámetro.

Cada IPC debe definir e implementar su propia estructura **channel\_t** para funcionar.

## FIFOs

Bla bla introducción, problemas encontrados, decisiones de implementación, /tmp/fifo /tmp/cliTEMPNAM. bla bla algún dibujito, bla

La estructura de **canal** en FIFOs tiene la siguiente forma:

**typedef** **struct** channel\_t {

**char** **\*** fifoIn;

**char** **\*** fifoOut;

**int** fdIn;

**int** fdOut;

} channel\_t;

Bla bla

## Message queues

Bla bla introducción, "en un comienzo se había pensado una estrategia de tener una cola por cada cliente bla bla bla bla, luego nos dimos cuenta que bla bla blaa bla", actualmente se usa una sóla cola encolando al mensaje con BLA BLA BLA.....decisión POSIX vs System V, problemas encontrados, algún dibujito, bla

La estructura de **canal** en las colas de mensaje tiene la siguiente forma:

**typedef** **struct** channel\_t {

Sem memory\_lock\_sem; *// Locks the shared memory so no data is corrupted*

Sem clitoserv\_sem; *// Signals server saying:"You have data to read"*

Sem servtocli\_sem; *// Signals client saying:"You have data to read"*

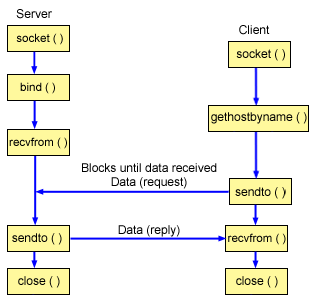
**void** **\*** buffer; *// Where client will read and write data*

} channel\_t;

Bla bla

## Sockets

Para la implementación de sockets se optó por usar el modelo **connectionless** o "no orientado a conexión" bajo el protocolo **UDP** *(SOCK\_DGRAM)*. Cabe destacarse que esta decisión no fue tomada desde un principio: al comenzar con el trabajo se había optado por el modelo **TCP** pero al darnos cuenta de que no era necesario hacer un doble *binding* ni que el servidor acepte las conexiones *accept()*, ni que el cliente las peticione *connect()*, decidimos cambiar al modelo **UDP.**



El servidor, a través de la función *upLink()*, crea un socket y le hace un *bind* al puerto 7000 y se quedará escuchando sobre él mediante *IPCListen()*. Este puerto es conocido por todos los clientes y es el lugar donde éstos peticionarán una nueva conexión. Por su parte, los clientes, al ejecutar *connectToServer()* no hacen más que enviar un mensaje de tipo *CONTACT* hacia el socket antes mencionado. Una vez recibido este mensaje, el socket creará y establecerá un canal exclusivo para el cliente a través de las funciones *createChannel()* y *establishChannel()*.

La estructura de **canal** en sockets tiene la siguiente forma:

**typedef** **struct** channel\_t {

**struct** sockaddr\_in **\*** client;

**int** port;

} channel\_t;

Contiene un puerto y una estructura *sockaddr\_in* que almacena todos los datos del cliente.

Es interesante destacar que, en el modelo no orientado a conexión, se pueden hacer uso de las funciones típicas de *send()* y *recv()*, pero también se nos provee de dos funciones nuevas llamadas *sendTo()* y *recvFrom()*. Lo buenos de utilizar estas últimas es que el *sendTo()* nos permite, por ejemplo, enviarle un mensaje al servidor sin antes haber realizado un *connect()*, *accept()* y demás cosas que necesita el protocolo TCP. Por su parte el *rcvFrom()* nos da la posibilidad de pasarle como parámetro un puntero a una estructura del tipo *sockaddr\_t* y, al recibir un mensaje, deja allí guardada toda la información del remitente. Esto hace que establecer el lazo entre el cliente y el servidor sea muy sencillo: el cliente simplemente le habla al puerto 7000, el servidor al usar *recvFrom()* obtiene su petición de conexión y todos sus datos, crea el socket para el nuevo cliente y le contesta a través de la dirección que guardó en el *recvFrom()* que la conexión se ha establecido y que desde ese momento en adelante el cliente debía hablarle y escuchar un determinado socket. El cliente hace los cambios internos y la conexión full-duplex ya queda conformada.

Se implementaron sockets de la familia **AF\_INET** por conveniencia y practicidad. Esta familia puede ser usada cuando los procesos a comunicar no están en la misma PC y la comunicación es por internet, pero al ser inclusiva, abarca también conexiones dentro del mismo ordenador.

Para finalizar, se implementó un manejador de señales con el fin de hacer una salida limpia al recibir una señal. Se cierran los *file descriptor* de los sockets abiertos a través de la función *close()* y recién luego de ésto se finaliza la ejecución.

## Shared Memory

La estrategia de implementación de la memoria compartida consiste en lo siguiente:

Se reserva un bloque de memoria compartida de tamaño fijo especificado en un *define (****SERVER\_BUF\_SIZE****)*, luego cada cliente ocupará una porción de ese ese bloque, definido en ***CLIENT\_BUF\_SIZE***. Lógicamente ***SERVER\_BUF\_SIZE*** es un múltiplo de ***CLIENT\_BUF\_SIZE*** y según por qué número se multiplique a este último, obtendremos la cantidad máxima de clientes que podrá soportar el servidor.

Por defecto el servidor aloca espacio para 1- clientes de **1 MB** cada uno. Siendo así, el cliente número 1 podrá escribir desde el offset *0* hasta el byte *1023*. El segundo cliente lo hará desde el *1024* hasta el *2047* y así sucesivamente.

Para que la comunicación entre los threads sea posible usando SHMM, se decidió utilizar 3 semáforos en el servidor y luego un par de semáforos por cada canal de conexión entre el cliente y el servidor. La justificación de esto es la siguiente:

* **SERVER\_SEM:** Este semáforo es el denominado "memory\_lock\_semaphore". Es el encargado de bloquear la zona de memoria compartida a la hora de escribir o leer de ella. Sirve para evitar datos inconsistentes y subsanar la presencia de condiciones.
* **SERVER\_CONTACT\_SEM:** Lo utilizan los clientes para solicitarle al servidor un nuevo pedido de conexión. El servidor principal hace un *wait()* sobre este semáforo y los clientes, al realizar *connectToServer()* lo señalizan.
* **RESPONSE\_CONTACT\_SEM:** Se utiliza para responderle al cliente que peticionó una nueva conexión. Debido a que la memoria compartida tiene un tamaño fijo, la cantidad de clientes que pueden conectarse al servidor está previamente calculada. En caso de que no haya lugar para un nuevo cliente, se le contesta al cliente que peticionó conexión que no hay más lugar disponible en el servidor; caso contrario se le informa el offset del bloque de memoria en el que deberá leer y escribir.

Lo interesante de utilizar esta estructura de semáforos es que el servidor logra no perder pedidos de conexión de ningún cliente. Se arma una suerte de **cola de pedidos** en la que los clientes que desean conectarse van aumentando el valor del *SERVER\_CONTACT\_SEM* y luego el servidor, a medida que tiene tiempo, los procesa y responde.

* **CLITOSERV\_SEM:** Una vez conectado, cada cliente tiene un semáforo exclusivo con el que le avisa al servidor que ha dejado nuevos datos en la zona compartida de memoria.
* **SERVTOCLI\_SEM:** Por el contrario del caso anterior, cuando el servidor desea avisarle al cliente que ha respondido a su petición, señaliza el *SERVTOCLI\_SEM.*

Se optó por utilizar **POSIX Semaphores** en vez de su versión de **System V** debido, casi completamente, a su facilidad para ser compartidos entre procesos. Al ser semáforos nombrados, basta con declararlos como globales en un archivo de header para que tanto el proceso cliente como el servidor puedan verlo. Esto evita la tediosa tarea de tener que compartir los semáforos a través del IPC, desperdiciando parte del bloque compartido de memoria para guardar los mismos. A su vez el uso de los semáforos POSIX resulta más sencillo que el de los System V: no debe crearse un arreglo de semáforos, no debe especificarse cuánto aumentar un semáforo al señalizarlo, etc. Se los crea con un determinado nombre y luego se utilizan las funciones wr *up()* y *down()* para aumentar o disminuír su valor en una unidad.

La estructura de **canal** en shared memory tiene la siguiente forma:

**typedef** **struct** channel\_t {

Sem memory\_lock\_sem; *// Locks the shared memory so no data is corrupted*

Sem clitoserv\_sem; *// Signals server saying:"You have data to read"*

Sem servtocli\_sem; *// Signals client saying:"You have data to read"*

**void** **\*** buffer; *// Where client will read and write data*

} channel\_t;

El semáforo para bloquear la zona compartida de memoria, los semáforos para sincronizar la comunicación con el servidor y un puntero a void indicando el buffer donde debe leer y escribir (el offset dentro del bloque compartido de memoria).

# Serialización de datos

Creemos de vital importancia redactar un pequeño apartado con el fin de explicar el procedimiento de serialización de datos:

La estrategia inicial para la serialización consistía en no tener estrategia. Cada IPC, en las funciones el servidor *IPCListen()* y *recvMessage()* en el cliente, debían armarse de su propia serialización. En líneas generales lo que se realizaba era un *switch()* gigante con diversos *cases*, un *case* para cada tipo de acción. Por ejemplo dentro del "*CASE LOGIN"* en el listen del servidor se resolvía todo el manejo de la deserialización del mensaje de login recuperando el nombre de usuario y la contraseña para así llenar los campos de la estructura *msg\_t* antes mencionada. En el cliente, para el mismo caso, se reservaba la memoria suficiente para todos los datos requeridos por el comando login y se armaba un *bytestream* con un formato similar a: [MSG\_SIZE MSG\_LOGIN userLen "username" passLen "password"]. Por último se hacía un memcpy() de todos los datos y recién allí se enviaba el mensaje.

Esta estrategia tiene varios puntos débiles:

* Se repite código en todos los IPCs para serializar los mensajes.
* No hay un formato estandarizado para cada tipo de mensaje: Sockets podía serializar el caso del *login* de una forma completamente distinta a FIFOs.
* Se debe utilizar un *switch()* gigantesco haciendo más ineficiente el proceso de serialización y deserialización de los mensajes (debe comparar contra cada uno de los aproximadamente 20 cases).
* No otorga un buen estilo de código.

Probado el funcionamiento de los IPCs bajo el método mencionado anteriormente, se decidió emprolijar el código y unificar la serialización de mensajes, delegando esta tarea en una única "clase". Esto nos permitereemplazar los grandes *switch()* por una única función d *serializeMsg()* y *deserializeMsg()* que internamente utiliza un **vector de punteros a función** accediendo de manera limpia e inmediata a la serialización necesaria, devolviendo el mensaje con todos sus campos completos, listo para ser utilizado.

# Draft

Es importa

# Interfaz de usuario

La idea del equipo, desde un principio, fue hacer uso de la biblioteca *ncurses* para poder proveerle al usuario una experiencia de juego mejor que simplemente una salida estándar. Con esta librería se podría hacer que el usuario vea un reloj con el tiempo restante para seleccionar un jugador y cosas de ese estilo. Debido a la falta de tiempo, no se logró hacer dicha implementación, pero sí se buscó generar una mejor interfaz de usuario mediante la utilización de colores en la salida.

Los errores salen impresos en la consola con color rojo, los comandos satisfactorios por su parte serán impresos en verde. A su vez, al listar las ligas, equipos y trades, se hizo uso de las tabulaciones que permite insertar la familia de funciones de *printf()* para así convertir el texto plano en un texto con cierto formato, indentado, formando tablas y agrupaciones.

# Conclusiones

La conclusión principal que puede sacarse de este trabajo es que **manejar memoria dinámica** en un desarrollo de software tan complejo y grande como este **requiere de un extremo cuidado.** A lo largo del desarrollo del trabajo práctico nos hemos encontrado con numerosas fallas en la aplicación, casi en un 100% producidas por un mal manejo de memoria. Si se reserva un *byte* menos para un *string*, si se intenta escribir algo de tamaño mayor al de un determinado buffer, si no se aloca de manera correcta una estructura, si se intenta desreferenciar un puntero que contiene basura, etc siempre caemos en fallas de segmentación o corrupción de la memoria. Creemos que este trabajo sirve puntapié inicial para concientizar al alumno sobre lo complicado que es manejar de manera correcta la memoria de un ordenador.

Como punto secundario pero no menos importante, es pertinente mencionar que el presente trabajo sirvió para familiarizarse con los conceptos de **IPCs**. Previo a la realización de éste el alumno desconocía casi completamente cómo es que hacen los procesos para comunicarse entre si. El trabajo aporta un gran caudal de información sobre las distintas formas de hacer efectiva esta comunicación y las ventajas y desventajas de cada una de ellas. No sólo se estudiaron en profundidad los distintos IPCs sino que, para algunos de ellos, se analizaron las distintas implementaciones propuestas por **POSIX** y/o **System V**, entendiendo que, según la aplicación a construir, resultará más conveniente el uso de uno u otra implementación.

Creemos que el desarrollo del trabajo deja bien claro las diferencias principales entre el uso de **threads** y **procesos**. Este es uno de los aprendizajes de mayor importancia, a nuestro entender, ya que las aplicaciones modernas reales explotan al máximo nivel estos conceptos y es más que bienvenido haber podido trabajar en ellos.

A su vez concluimos que, así como es importante tener un buen manejo de la memoria, también es un requerimiento fundamental tener bien presente el concepto de **condiciones de carrera** y **servidores concurrentes**.Se destaca el aprendizaje de los monitores, mutexs y semáforos para lograr una comunicación sincronizada y lograr datos consistentes en la memoria.

Para finalizar y ya pasando al plano humano, opinamos que el TP sirvió para afrontar la problemática real de construir un sistema complejo **en equipo**. Dentro de la carrera es la primer experiencia que el alumno tiene que lidiar condesarrollo de considerable envergadura en el que el **factor humano** empieza a tener importancia ya que las tareas deben ser distribuidas y delegadas en pos de completar satisfactoriamente el objetivo común. No sólo las tareas deben ser repartidas sino que también se deben **tomar decisiones** y acordar cómo se hará cada cosa, hecho que puede provocar disconformidades o roces. De todas formas el equipo pudo solucionar esta problemática a lo largo del desarrollo del trabajo