

Universidad Tecnológica Nacional – Facultad Regional Buenos Aires

Ingeniería en Sistemas de Información

Sistemas Operativos (95-2027)

**Trabajo Práctico Nro. 2 – 2do Cuatrimestre 2009**

**Sistema de Archivos Distribuido**

**Revisión 0.44**

**Índice**

1.- Introducción 3

2.- Objetivos del Trabajo Práctico 3

3.- Características del Sistema Distribuido 4

4.- Descripción detallada de las entregas 13

4.1.- Primer entrega 13

4.2.- Segunda entrega 15

4.3.- Tercer entrega 18

4.4.- Cuarta entrega 19

4.5.- Quinta entrega 24

5.- Requerimientos tecnológicos y limitaciones 27

6.- Anexo – Log y Debugging 30

7.- Anexo – Documentación 32

8.- Anexo – Protocolos de comunicación 34

8.1.-Inter. Process Comunications (IPCs) – Protocol IRC/IPC standard 34

8.2- Handshake 34

**1.-Introducción**

El trabajo práctico de este cuatrimestre consta de realizar un servidor de FTP soportado por un kernel remoto encargado de manejar un file system distribuido. El conjunto formado por el servidor FTP, el kernel y el file system conformarán un modelo de computadora simplificada. A esta computadora simplificada se le conectarán diversos tipos de dispositivos de almacenamiento, ya sean discos o arrays de discos.

Uno de los objetivos del trabajo práctico es mostrar las distintas capas que componen en este caso un proceso de almacenamiento, su independencia y su interdependencia, pero usando diseño aplicado en muchas operaciones del sistema operativo.

Dada la complejidad, el mismo se dividió en entregas parciales las cuales permiten diseñar la arquitectura definitiva, de forma gradual, simplificando su desarrollo. El trabajo práctico consta de varios procesos divididos en tres categorías: Aplicación, Sistema Operativo y Almacenamiento, los cuales se van a comunicar entre sí mediante el paso de mensajes mediante sockets TCP/IP.

Se enumeran a continuación los conceptos teóricos y prácticos más significativos sobre sistemas operativos que cubre el trabajo práctico y que el alumno aprenderá y podrá relacionar con la teoría:

* Creación y manipulación de procesos y threads mediante la API del sistema.
* Sincronización de Procesos, IPC y manejo de señales.
* Administración de Memoria.
* Manejo del file system a través de la interfaz de sistema.
* Introducción a los Sistemas Distribuidos.
* Arquitecturas basadas en capas físicas, protocolos y mensajería.
* Diferencias entre las plataformas más populares del mercado.

**2.-Objetivos del Trabajo Práctico**

Desde el punto de vista académico el trabajo está diseñado para:

* Que los alumnos adquieran los conocimientos prácticos del uso de un conjunto de herramientas que ofrecen los sistemas operativos modernos.
* Que entiendan la importancia de una norma o protocolo estándar en la comunicación entre procesos y diferentes plataformas.
* Que dominen los problemas específicos de este tipo de implementaciones.
* Que apliquen en forma práctica el uso de lenguaje C en implementaciones de bajo nivel.
* Que el grupo de alumnos aprendan el trabajo en equipo, las problemáticas y las responsabilidades que eso implica.

**3.- Características del Sistema de archivos distribuido.**

A continuación se hace una breve introducción a cada uno de los componentes que intervienen en el sistema. El siguiente diagrama de distribución física muestra en forma resumida los principales componentes.

**Diagrama de distribución física**

VSK

SFS

Server FTP

IPC/IRC sobre TCP/IP

Cliente FTP

Cliente FTP

Cliente FTP

FTP sobre TCP/IP

IPC/IRC sobre TCP/IP

Computadora Simplificada

Array1

DISCO A

DISCO B

IPC/IRC sobre TCP/IP

DISCO 1

IPC/IRC sobre TCP/IP

**Nivel de Aplicación - Windows**

**Servidor FTP**

En esta capa se encuentra el servidor de protocolo de transferencia de archivos o FTP por su sigla en inglés (File Transfer Protocol - <http://www.rfc-es.org/rfc/rfc0959-es.txt>). Proveerá servicios de transferencias de archivos usando el protocolo ftp estándar hacia el exterior. Internamente será la aplicación que haga uso de los servicios proporcionados por el sistema operativo virtual.

**Aspectos de diseño:**

La aplicación se desarrollará para la plataforma **Windows** y será **multithread** para aprovechar al máximo las capacidades de los sistemas multiprocesadores actuales.

Internamente esta aplicación está desarrollada con una característica particular, ya que utiliza una API (application programming interface) para acceder a los servicios de un Kernel que estará funcionando en otra computadora dentro de la red.

**Capa de Sistema Operativo - Linux**

**VSK – *Very Simplified Kernel***

Este proceso representará un Kernel muy simplificado el cual solo expondrá servicios para comunicarse con el file system. Esta aplicación distribuida se ejecutará remotamente y expondrá una serie de system calls para ofrecer dichos servicios de acceso.

**VSK API**



La capa de aplicación no accederá directamente a las system calls remotas del kernel. Deberán desarrollar una biblioteca de enlace dinámico la cual expondrá una API que encapsulará los detalles de las llamadas a las funciones. Esto permite una separación de capas más limpia y permitirá cambiar los detalles de su implementación sin afectar a la capa de aplicación. En el diagrama de procesos se detallan los componentes y su analogía con la separación de capas que existen entre la aplicación y el sistema operativo. Entonces, cualquier aplicación que desee utilizar este kernel deberá utilizar esta biblioteca.

**Aspectos de Diseño**

El Kernel será un proceso desarrollado para la plataforma **Linux** y se encargará de varias tareas específicas. Será el encargado de manejar las system calls para el manejo de file system. El proceso estará a la espera de un mensaje solicitando un servicio, este detectará el evento y llamará al procedimiento que atiende esa solicitud.

El primer proceso que se iniciará en el TP, una vez finalizado su desarrollo, es el proceso “Kernel (VSK)” el cuál quedará a la espera de conexiones. A este proceso se le conectará un proceso “Simple File System (SFS)” y un proceso “Servidor FTP”. La unión de estos 3 procesos conformará lo que conceptualmente se denomina una “Computadora Simple (SC)”. A esta “Computadora Simple (SC)” se le conectarán indistintamente discos y arrays.

Estos discos y arrays se conectarán al proceso “Kernel (VSK)”.

Dado que todos los procesos se conectarán al mismo puerto TCP del proceso “Kernel (VSK)” será necesario que dichas conexiones se discriminen mediante un handshake (ver apartado handshake).

Desde la consola del proceso “Kernel (VSK)”, la “Computadora Simple (SC)” tendrá la capacidad de montar y desmontar discos y arrays denominados genéricamente dispositivos de almacenamiento o DAs.

**Caché de File System**

El proceso “Kernel (VSK) contará con una cache de file system para poder acceder a sectores sin necesidad de acceder a los dispositivos de almacenamiento.

**Comportamientos definidos**

Al iniciar el proceso, este obtendrá las siguientes características de un archivo de configuración:

* + Puerto TCP en el que escuchará peticiones
  + Dirección IP, puerto, usuario administrador y password del OpenDS

**Montaje y formateo de los dispositivos de almacenamiento (DA):**

El Kernel recibirá el identificador único del DA y chequeará que actualmente no esté en uso. Si no lo está entonces se comunicará con el file system preguntando si posee un formato para dicho DA, en otras palabras si posee una tabla de sectores libres y una tabla de archivos.

En caso afirmativo, se concluirá el montaje y los datos del disco estarán a disposición de las aplicaciones. Caso contrario se solicitará por pantalla que se formatee dicho dispositivo.

El montaje se realizará siguiendo una estructura de directorios donde el directorio raíz es “/”, cada DA es montado como un directorio separado, el separador de directorios es “/” y no existen las estructuras de subdirectorios.

**Proceso de formateo:**

Se le solicitará al DA su información física *CHS* (*Cylinders, Heads and Sectors*).

Con dicha información el Kernel calculará la cantidad de sectores totales y la capacidad máxima de almacenamiento. Luego le solicitará al SFS que cree la tabla de sectores para ese disco.

La estructura general del file system se implementará en opends, accedida a través del proceso SFS

**Manejo de archivos**

El “proceso Kernel (VSK)” manejará una tabla de archivos llamada open-files table (OFT) la cual contiene información de todos los archivos abiertos. Cuando una operación sobre un determinado archivo es solicitada, el sistema buscará el índice (descriptor) en la tabla y no será necesario acceder al sistema de archivos. Esta tabla será implementada en memoria del proceso VSK (**Linux**).

**Nombres de archivo**

El nombre completo de un archivo se especifica de la siguiente forma:

/[DA]/[nombre de archivo]

Donde [DA] refiere al nombre del dispositivo de almacenamiento y [nombre de archivo] al nombre de dicho archivo el cual puede ser una secuencia de hasta 30 caracteres combinando alfanuméricos y el carácter punto.

**Ejemplo:**

/discoA/PassWord99.txt

Nota: Tanto los nombres de los archivos y como los nombres de los dispositivos de almacenamiento deberán ser CASE-SENSITIVE (sensibles a mayúsculas).

**OFT**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Descriptor único (int)** | **Nombre del DA (char)** | **Nombre de archivo (char)** | **Modo de apertura (int)** | **Tamaño total bytes (long)** | **Buffer (1KB)** | **Sectores que componen el archivo (struct)** |
| 5 | discoA | passwd | 0 | 1450 | {contenido del archivo} | {lista de número de sectores que componen el archivo} |

**System Calls del VSK**

El kernel pondrá a disposición de las aplicaciones las siguientes system calls las cuales podrán ser accedidas mediante el paso de mensajes utilizando el protocolo IPC/IRC

sys\_open(): Abre un archivo existente.

sys\_read() Lee un flujo de 1 KByte de un archivo.

sys\_write() Escribe un flujo de 1 KByte en un archivo.

sys\_close() Cierra un archivo abierto.

sys\_flush() Vuelca el contenido del buffer de escritura al disco.

sys\_flist() lista un directorio del vsk.

**SFS – Simplified File System**

Este proceso será el encargado de asociar un nombre de archivo con los correspondientes bloques que lo componen. El file system deberá permitir almacenar el propietario, los permisos y el tamaño del archivo. Por simplicidad el SFS no soporta estructuras de **directorios**

**Aspectos de diseño:**

Estará desarrollado para la plataforma Linux. Como soporte de datos utilizará **OpenDS** y el protocolo **LDAP** dada la simplicidad de esta herramienta para asociar varios elementos a una clave.

Como todos los procesos, el SFS, proveerá, una API con la cual se podrá comunicar con el VSK mediante el paso de mensajes utilizando el protocolo IPC/IRC y tendrá una consola que permitirá hacer uso de la API mediante comandos definidos.

**Comportamientos definidos**

Al iniciar el proceso este obtendrá las siguientes características de un archivo de configuración:

* + Dirección IP y puerto TCP del VSK
  + Dirección IP, puerto, usuario administrador y password del OpenDS

Se conectará a ambos procesos y quedará a la espera de instrucciones.

. La estructura general del SFS tiene la siguiente forma:

SFS

**ARRAY 1**

archivo1

archivo2

password.txt

1

2

5

11

12

Owner: user

Perms: rwx---r--

Tamaño: 1200 bytes

Owner: root

Perms: rwx---r--

Tamaño: 700 bytes

Owner: root

Perms: rwx------

Tamaño: 1024 bytes

13

6

**ARRAY B**

**ARCHIVOS**

8

**SECTORES LIBRES**

3

4

7

9

10

14

15

…

124

125

file.doc

8

Owner: root

Perms: rwx------

Tamaño: 1024 bytes

10

**SECTORES LIBRES**

1

2

3

4

6

9

11

…

998

999

**ARCHIVOS**

5

7

**Caché de file system**

El proceso VSK a su vez tendrá acceso a una cache de sectores, la misma permitirá acceder a sectores del archivo desde el repositorio de caching (opends). Esta cache se deberá poder activarse y desactivarse por archivo de configuración. La estructura de la cache será de la siguiente forma:

|  |  |  |
| --- | --- | --- |
| Dispositivo | idSector1 | Sector1 |
| Disco1 | 0x1 | .... |

El VSK cada vez que quiera leer un archivo irá a buscar la lista de sectores al SFS (mediante protocolo IPC/IRC), a continuación el VSK preguntará al opends si esos par de sectores se encuentran en la cache. Si existen, el VSK buscará dichos sectores en la cache, si el cache encuentra el bloque de sectores (cache hit) los devolverá, sino (cache miss) el VSK irá a buscar el par de sectores al dispositivo de almacenamiento correspondiente, actualizará la cache y los devolverá al server ftp.

La política de caching del FSF será la de LRU (Least Recently Used). Esto quiere decir que la cache tendrá un tamaño máximo de sectores configurable por archivo, y en el proceso de almacenamiento de sectores, la selección del próximo sector a reemplazar será el que resulte de aplicar esta política. De igual manera, cuando un sector ó grupo de sectores son escritos a un DA el VSK deberá anular dicho grupo de la cache ya que su información estaría desactualizada generando inconsistencias.

El VSK deberá escribir en el archivo log cada uno de los “cache hit” y “cache miss” de sectores generados en la cache para el DA y el archivo correspondiente.

**Nivel de Almacenamiento**

**Disco**

Este proceso simulará un disco rígido real. Tendrá como características sus valores de CHS, retardo rotacional y posición actual del cabezal de lectura.

Utilizará OpenDS para almacenar los sectores de datos de 512 Bytes de forma que los mismos puedan persistir una vez dado de baja el proceso y los organizará identificándolos mediante el número de sector iniciando por el cero. Contará con una cache que será implementada a través de un proceso encargado de hacer de interface con memcached (<http://www.danga.com/memcached/>). Por simplicidad, los discos tendrán un solo plato, por ende un solo cabezal. El algoritmo de posicionamiento del cabezal será C-SCAN.

1

2

3

0

5

6

7

4

9

10

11

8

La figura muestra la disposición de los sectores en caso de tratarse de un disco de 1 cabezal, 3 cilindros y 4 sectores por pista: CHS (1, 3, 4)

**Aspectos de diseño:**

Este proceso será desarrollado bajo Linux. Al iniciar, verificará en su archivo de configuración si tiene la cache activada. En dicho caso creará el proceso hijo “cache” al cual se comunicará mediante IPC.

Como todos los procesos, el disco, proveerá, una API definida por el alumno con la cual se podrá comunicar indistintamente con un Array o con un Kernel mediante el paso de mensajes utilizando el protocolo IPC/IRC y tendrá una consola que permitirá hacer uso de la API mediante comandos definidos.

Es importante recalcar que si bien el proceso simula un disco real dados sus parámetros físicos, esta información será únicamente mostrada en el archivo log del proceso. De ninguna manera se pretende que el tiempo de respuesta de los mensajes del proceso coincida con los tiempos calculados y grabados en el archivo log del proceso.

Intentará conectarse al dispositivo de destino, el cual puede ser uno de los siguientes y quedará a la espera de instrucciones.

**Comportamientos definidos**

Al iniciar el proceso este obtendrá las siguientes características de un archivo de configuración:

* + Nombre (deberá ser único en el sistema)
  + Dirección IP y puerto TCP del proceso al que se conectará
  + CHS. La cantidad de sectores debe ser siempre par.
  + Revoluciones por minuto
  + Cache (si/no)

Luego calculará y mostrará en el archivo log del proceso:

* Tiempo medio y peor tiempo de búsqueda de un sector
* Tasa de transferencia
* Capacidad total del disco
* Cantidad de sectores
* Tiempo medio de acceso de la cache

Intentará conectarse al dispositivo de destino, el cual puede ser uno de los siguientes y quedará a la espera de instrucciones.

- Un VSK

- Un Array

**Caché de almacenamiento**

Al disco se le podrá habilitar un proceso de cache, el cual almacenará la información de los sectores solicitados en memcached (<http://www.danga.com/memcached/>). Ester proceso se conectará al disco mediante IPC.

En caso de solicitarse un sector y que el mismo se encuentre en cache, se deberá responder la información almacenada en la misma (Cache HIT), en caso contrario deberá reportar que dicha información no está disponible en la cache en el archivo log del proceso (Cache MISS) y se continuará el proceso de búsqueda habitual en el disco.

Si se escribiera un sector en el disco cuya información se encuentra actualmente en cache, dicho registro deberá ser actualizado.

**Comportamientos definidos**

Al iniciar el proceso este obtendrá las siguientes características de un archivo de configuración:

* + Nombre (deberá ser único en el sistema)
  + IP y Puerto TCP de 2 servidores memcached
  + Dirección IP y puerto TCP del proceso al que se conectará

**Array**

Este proceso simulará una controladora de RAID 1 la cual se encarga de tener la información de los “n” discos conectados a ella de forma espejada, reenviando de forma eficiente las solicitudes que recibe del VSK.

**Anexo RAID 1**

Una controladora RAID 1, al recibir la instrucción de escribir un sector, la reenvía a todos los discos que componen dicho array, logrando de esta forma que todos los discos tengan en todos sus sectores la misma información.

Dado este estado de sincronía absoluta, en caso de tener que leer información, puede delegar las solicitudes en paralelo a cualquiera de los discos que lo componen, ya que como dijimos anteriormente, todos tienen la misma información, por ende cualquiera pueden servir dicha solicitud.



ARRAY

1

VSK

DISCO 3

DISCO 2

DISCO 1

putSector(8: [512 b], 9:[512 b]);

putSector(8: [512 b], 9:[512 b]);

putSector(8: [512 b], 9:[512 b]);

putSector(8: [512 b], 9:[512 b]);

La figura muestra como el Array replica los pedidos de escritura a cada uno de los discos.

De esta **forma los sectores de todos los discos almacenan los mismos datos**.

Esto genera un beneficio muy significativo en el tiempo de lectura, sin, prácticamente, aumentar el tiempo de escritura.

La desventaja de RAID 1 radica en el costo, dado que el espacio de discos completos es desaprovechado en favor de la redundancia. RAID 1 generalmente se implementa con dos disco en espejo, pero en realidad se puede duplicar tantas veces como discos estén conectados al RAID.

El RAID 1 además incrementa notablemente la tolerancia a fallos del sistema de almacenamiento ya que en caso de que un disco falle, el resto mantiene disponible la información.

**Aspectos de diseño:**

Este proceso será desarrollado bajo Solaris y creará un thread por cada disco que se le conecte. Será el encargado de balancear la carga de las solicitudes equitativamente de forma circular entre todos los dispositivos que tenga conectado.

**Comportamientos definidos**

Leerá de archivo de configuración al menos los siguientes parámetros:

* Nombre (debe ser único en el sistema)
* Dirección IP y puerto TCP del proceso al que se conectará
* Cantidad de discos que componen el array
* Nombres de los discos que componen el array

Permanecerá fuera de servicio a la espera de las conexiones de los discos que lo componen. Una vez completo modificará su estado a: inicializado e intentará conectarse al dispositivo de destino, el cual puede ser uno de los siguientes.

- Un VSK

Al iniciarse el proceso mostrará en el sistema:

* Cantidad y nombres de los discos que lo componen
* Capacidad total del Array

Durante todo el proceso el sistema mostrará:

* Discos actualmente conectados y sus estados
* Estado del array
  + **Inicializado**

En este estado el array podrá responder solicitudes normalmente intentando balancear la lectura entre sus discos en sincronía.

Consta de los siguientes sub-estados:

* + - **Sincronizado**: Estado ideal donde todos los discos se encuentran en sincronía
    - **Sincronizando**: Un disco está siendo actualizado. En caso de recibir una operación de escritura, la misma deberá procesarse también en el disco que está siendo sincronizado, solo si dicho número de sector ya ha sido copiado del disco de origen (y por ende está siendo modificado)
  + **Fuera de servicio**

En este estado el array no podrá responder solicitudes y notificará al VSK mediante un mensaje de “Fuera de servicio”.

Consta de los siguientes sub-estados:

* + - **Incompleto**: Algunos de los discos especificados por archivo de configuración no se conectó con el array.
    - **Sincronizado**: El array está funcional pero fue puesto fuera de servicio por consola.
    - **Corrupto**: Ninguno de los discos que actualmente componen llegaron a sincronizarse.

**4.-Descripción detallada de las entregas**

Es importante recalcar que si bien las entregas tienen sus fechas de entrega y están numeradas y ordenadas existe mucho trabajo que correctamente dividido entre los integrantes del grupo se puede realizar de forma paralela.

**4.1.-Primer entrega**

**Descripción**

En esta entrega se deberá desarrollar la primera versión del servidor FTP. Se iniciará por línea de comando y tomará los parámetros de un archivo de configuración (que se encontrará en mismo directorio donde corre la aplicación).

Una vez iniciado, aceptará las conexiones de clientes sin importar su usuario o password (modo anónimo) y negociará con el cliente el modo pasivo. Una vez establecida la conexión desde el cliente creará un thread para cada cliente para manejar la conexión de comandos y otro thread para la conexión de datos (dos threads por cliente) según el protocolo ftp. En esta entrega el servidor compartirá la información de un directorio local de la computadora configurado por archivo.

RFC: <http://es.wikipedia.org/wiki/File_Transfer_Protocol>

**Comportamientos definidos**

Al iniciar el proceso este obtendrá las siguientes características de un archivo de configuración:

* + Puerto TCP en el que escuchará peticiones FTP

El servidor aceptará conexiones de clientes de forma anónima y funcionará en el modo pasivo.

Deberá soportar al menos los siguientes comandos

|  |  |
| --- | --- |
| **Pwd** | Muestra el directorio activo en el servidor |
| **cd** *directorio* | Cambia el directorio de trabajo en el servidor |
| **Ls** | Muestra el contenido del directorio en el que estamos en el servidor |
| **delete** *archivo* | Borra un archivo en el servidor. |
| **get** *archivo* | Obtiene un archivo. |
| **put** *archivo* | Envía el archivo indicado al directorio activo del servidor. |
| **?** o **help** | Muestra una lista de los comandos disponibles |
| **noop** | *(No Operation)* Se le comunica al servidor que el cliente está en modo de no operación, el servidor usualmente responde con un "ZZZ" y refresca el contador de tiempo inactivo del usuario. |
| **pasv / port** | Comandos necesarios para iniciar la comunicación de modo pasivo. |

*Multithreading*

Ante la solicitud de un nuevo archivo, la aplicación creará un nuevo thread de datos para atender la solicitud. Adicionalmente detectará si está corriendo sobre un sistema de múltiple procesadores y cambiará de forma equitativa la afinidad de un thread a un determinado procesador. En caso de que la aplicación detecte un sistema mono procesador esta característica no debería estar disponible. Para el manejo de threads ver la sección de requerimientos tecnológicos. El siguiente diagrama muestra la asignación de threads en una plataforma con dos procesadores:



*File system*

La aplicación utilizará las funciones para manejo de file system que brinda la Windows API, como hace referencia la sección File system del anexo Requerimientos tecnológicos. Es importante tener en cuenta que dichas funciones serán modificadas para luego interactuar con el sistema de archivos desarrollado por el alumno por lo que se recomienda modularizar los procesos de lectura y escritura de archivos.

|  |  |
| --- | --- |
| **Modo de testeo de la entrega** | |
| * Se recomienda verificar la integridad de los archivos transferidos en ambos extremos con el comando md5sum (en Windows deberá usarse un comando similar). * Controlar que el uso de CPU corresponda con el procesador correcto en caso de correr sobre un sistema multiprocesador. |

|  |  |
| --- | --- |
| **Tiempo estimado** | dos semanas |
| **Tipo de entrega** | No Obligatoria |
| **Fecha** | 26/09/2009 |

**4.2.-Segunda entrega**

**Nota: Se recomienda al alumno leer los capítulos 10, 11, 12, 13 del libro Silberschatz correspondientes a la parte cuatro, administración de almacenamiento.**

**Descripción**

En esta entrega se desarrollarán los procesos disco y array. Es importante recalcar que los mensajes que reciben y envían ambos procesos son idénticos, lo que permite generar estructuras de RAID1 interconectando estos componentes en forma de árbol.

**Primitivas del Disco**

El Disco pondrá a disposición de un VSK o de un Array las siguientes operaciones las cuales podrán ser accedidas mediante el paso de mensajes utilizando el protocolo IPC/IRC

**getSectores():** Dado dos números de sectores, este proceso, antes de devolver dichos sector al solicitante, calculará e imprimirá en el log:

* el tiempo que tardaría en servir dicho requerimiento un disco rígido real
* los sectores que tendría que atravesar antes de servirlos
* la posición actual del cabezal
* promedio de cache hit/miss

Devuelve 1024 bytes donde los primeros 512 Bytes corresponden al primer sector solicitado y los siguientes al segundo. En caso de recibir la solicitud de sector “null”, la misma deberá ser ignorada y responder con 512 bytes en ‘\0’

**putSectores():** Dada una estructura de escritura de 1032 bytes, antes de escribir los sectores y notificar el éxito de la operación al solicitante, el proceso calculará y escribirá en el archivo log del proceso:

* el tiempo que tardaría en escribir dicho sector
* los sectores que tuvo que atravesar antes de escribirlo
* la posición actual del cabezal

En el caso que el sector existiera en el repositorio (alocado previamente por un archivo actualmente no existente), se deberá actualizar con el contenido del archivo actual.

**Estructura de escritura:**

Son 1032 bytes que corresponden a:

|  |  |  |  |
| --- | --- | --- | --- |
| Id sector 1 (4 Bytes) | Buffer sector 1 (512 Bytes) | Id sector 2 (4 Bytes) | Buffer sector 2 (512 Bytes) |

**getCHS():** Devuelve la información física obtenida de archivo de configuración.

**Consola**

**posicionCabezal():** Imprimirá en pantalla la posición actual del cabezal

**obtenerSectores([lista sectores])**: recibirá como parámetro una lista de no más de 20 sectores y deberá mostrar en pantalla por cada sector la información descripta en los comportamientos definidos junto con los primero 10 y los últimos 10 caracteres de dicho sector.

Ejemplo: obtenerSectores(100, 101, 102)

Posición actual: 0

Sectores Leídos: 21, 32, 43, 54, 65, 76, 87, 98, 99, 100

Tiempo consumido: 16ms

Sector 100: “Información”…”del final.”

Sectores Leídos: 101

Tiempo consumido: 2ms

Sector 101: “primeros “…”ultimos 10 “

Sectores Leídos: 102 – En cache

Tiempo consumido: 0.1ms

Sector 102: “aabbccddee”… “1234567890”

**escribirSector([numero de sector], [buffer de hasta 512 caracteres]):** Imprimirá en pantalla la misma información que el comando obtenerSectores() sólo que en este caso escribirá el sector pasado como parámetro en el buffer.

**Primitivas del Array**

El array pondrá a disposición de un VSK las siguientes operaciones las cuales podrán ser accedidas mediante el paso de mensajes utilizando el protocolo IPC/IRC

**getSectores():** Dado dos números de sectores este proceso deberá balancear de a una la solicitudes mediante round robin a los DAs asociados al array. Round robin funciona como una cola circular, a medida que se hace una solicitud, el nodo que se encuentra al principio (disco1), una vez consultado, pasa a estar en la última posición de la cola y así sucesivamente.

Devuelve 1024 bytes donde los primeros 512 bytes corresponden al primer sector solicitado y los siguientes al segundo. En caso de recibir la solicitud de sector “null”, la misma deberá ser ignorada y responder con 512 bytes en ‘\0’.

Ejemplo:

ARRAY

1

VSK

DISCO 3

DISCO 2

DISCO 1

getSectores(8, 9);

getSectores(9, null)

getSectores(8, null)

**putSectores():** Dada una estructura de escritura de 1032 bytes (ver primitivas del disco), deberá reenviar dicho comando a todos los DAs asociados al array.

**getCHS():** Devuelve la información física que coincide con todos los dispositivos del array.

**Consola**

Una vez inicializado el array el proceso deberá permitir:

* **Agregar DA al array**:
  + Al recibir una conexión de un DA el array deberá corroborar que el mismo sea idéntico en características a los demás dispositivos que componen el array. En caso de serlo deberá permitir sincronizar dicho dispositivo sin que el array salga de servicio.   
    La operación no podrá ejecutarse si el array se encuentra actualmente en proceso de sincronización o fuera de servicio. Deberá mostrar el progreso en pantalla.
* **Eliminar DA del array**:
  + Esta operación no podrá realizarse si al eliminar un disco, los discos que quedan en el array no se encuentran sincronizados.
* **Poner fuera de servicio/Reanudar servicio**:
  + No se podrá poner fuera de servicio un array en proceso de sincronización.

**Proceso de sincronización:**

Dado que el array conoce la cantidad de sectores que componen sus discos, haciendo uso de las primitivas, deberá copiar los datos, de a dos sectores, almacenados en algún disco en sincronía a aquel disco que requiera sincronización.

**ARRAY A**

**DISCO 1**

**DISCO**

**Nuevo**

Datos = **getSectores**(83, 84);

**putSectores**(struct Datos);

|  |  |
| --- | --- |
| **Modo de testeo de la entrega** | |
| * Mediante la consola leer y escribir bloques en los dispositivos de almacenamiento * Desconectar y conectar discos al array para observar la correcta sincronización. |

|  |  |
| --- | --- |
| **Tiempo estimado** | Tres semanas |
| **Tipo de entrega** | No Obligatoria |
| **Fecha** | 17/10/2009 |

**4.3.-Tercer entrega**

**Descripción**

En esta entrega se creará el proceso VSK. Aceptará conexiones de los diversos dispositivos de almacenamiento y podrá leer y escribir sectores de los dispositivos de almacenamiento utilizando el api provisto por los mismos. Agrupándolos en el buffer de la OFT en bloques de 1KB. Desde la consola, el VSK deberá poder ejecutar operaciones get\_sectores(), put\_sectores() y get\_CHS, imprimiendo los resultados en pantalla.

Las systems calls que el proceso vsk podrá invocar desde los dispositivos de almacenamiento son:

get\_sectores();

put\_sectores();

get\_CHS();

También se deberá implementar el proceso de cache en los dispositivos de almacenamiento, comunicándolo por IPC al dispositivo de almacenamiento correspondiente. Este proceso deberá utilizar la tecnología memcached para guardar los sectores en memoria.

|  |  |
| --- | --- |
| **Modo de testeo de la entrega** | |
| * Conectar y desconectar DAs. * Desde la consola leer/escribir sectores en los discos utilizando el buffer intermedio de la OFT. |

|  |  |
| --- | --- |
| **Tiempo estimado** | Tres semanas |
| **Tipo de entrega** | Obligatoria |
| **Fecha** | 07/11/2009 |

**4.4.-Cuarta entrega**

**Descripción**

A la estructura actual se debe agregar el SFS. El mismo se conectará al VSK y liberará rutinas para ordenar lógicamente los sectores en archivos. El VSK permitirá formatear discos que al montarse no tengan formato e implementará las sys\_calls para acceder a los archivos. En esta entrega se deberá incluir la cache del file system del VSK que es una cache de sectores por DA implementada con opends.

**Systems Calls del VSK:**

**Operación sys\_open()**

Esta operación será utilizada para abrir archivos en modo lectura, escritura o para eliminar archivos

**Parámetros**

**Tipo de apertura**: *LECTURA, ESCRITURA o ELIMINAR*

**Nombre del archivo completo**: /[DA]/[nombre de archivo]

**El VSK realizará al menos las siguientes operaciones:**

1. Verificará que el archivo no esté actualmente abierto.
2. Consultará con el SFS que el archivo exista. Primitiva del SFS: existeArchivo().

En caso de abrir para lectura:

3) Solicitará al SFS la lista de bloques que componen dicho archivo. Primitiva del SFS: infoArchivo().

4) Creará el registro de la OFT y almacenará dicha lista en el campo de sectores.

En caso de abrir para escritura:

3) En caso de existir el archivo, solicitará al SFS que lo elimine. Primitiva del SFS: eliminarArchivo().

4) Solicitará al SFS que cree el archivo. Primitiva crearArchivo().

5) Creará el registro de la OFT.

En caso de abrir para eliminar:

* 1. Creará el registro de la OFT

Por último responderá el éxito de la operación o su correspondiente mensaje de error el cual también será impreso en el archivo log del proceso.

**Operación sys\_read ()**

Esta operación permite leer un bloque de un determinado descriptor de archivo abierto para lectura.

Es importante recordar que el kernel trabaja con bloques fijos de **1** KB.

**Parámetros**

**Descriptor de archivo:** Número identificador del descriptor a leer.

**El VSK realizará al menos las siguientes operaciones:**

1. Verificará en la OFT que dicho descriptor corresponda a un archivo abierto para lectura.
2. Obtendrá de la lista de sectores de la OFT los dos próximos sectores a leer y los solicitará al dispositivo de almacenamiento correspondiente utilizando la primitiva getSectores(). Temporalmente utilizará el buffer de 1KB como memoria intermedia.  
   Nota: En caso que la cantidad de sectores a leer sea menor a dos, o alguno de dichos sectores se deba leer parcialmente (dado el tamaño máximo del archivo) el buffer de 1KB deberá ser rellenado con caracteres ‘\0’.
3. Enviará el buffer correspondiente al solicitante junto con un valor correspondiente al éxito de la operación o su correspondiente mensaje de error el cual también será mostrado por pantalla.

**Operación sys\_write()**

Esta operación permite escribir un bloque de un determinado descriptor de archivo abierto para escritura. Es importante recordar que el kernel trabaja con bloques fijos de **1** KB.

**Parámetros**

**Descriptor de archivo:** Número identificador del descriptor a escribir.

**Buffer**: Bloque de hasta 1KB a **escribir** en un archivo.

**El VSK realizará al menos las siguientes operaciones:**

1. Verificará en la OFT que dicho descriptor corresponda a un archivo abierto para escritura.
2. Escribirá en el buffer de la OFT el buffer recibido rellenado hasta 1KB en caso necesario.
3. Ejecutará la operación sys\_flush().
4. Actualizará el tamaño del archivo en el SPS.
5. Responderá el éxito de la operación al solicitante o su correspondiente mensaje de error el cual también será mostrado por pantalla.

**Operación sys\_flush ()**

Esta operación persiste en el correspondiente dispositivo de almacenamiento el buffer acumulado para dicho archivo.

Si bien forma parte de los system calls del VSK no está liberada para el uso de las aplicaciones.

**Parámetros**

**Descriptor de archivo:** Número identificador del descriptor a escribir.

**El VSK realizará al menos las siguientes operaciones:**

1. Verificará en la OFT que dicho descriptor corresponda a un archivo abierto para escritura.
2. Solicitará al SFS dos sectores libres del DA correspondiente. Primitiva: dosSectoresLibres().
3. Escribirá ambos sectores del correspondiente DA utilizando la primitiva: putSectores().
4. Responderá el éxito de la operación o su correspondiente mensaje de error el cual también será mostrado por pantalla.

**Operación sys\_close ()**

Esta operación es la forma de concluir una transacción sobre un archivo y registrar el éxito de la misma.

**Parámetros**

**Descriptor de archivo:** Número identificador del descriptor a cerrar.

1. Verificará en la OFT que dicho descriptor corresponda a un archivo abierto.

En caso de escritura:

2) Enviará al SFS el tamaño final del archivo. Primitiva: actualizarTamanio().

En caso de eliminar:

2) Solicitará al SFS que elimine el archivo. Primitiva: eliminarArchivo().

Eliminará el registro correspondiente de la OFT y confirmará el éxito de la operación o su correspondiente mensaje de error el cual también será impreso en el archivo log del proceso.

**Operación sys\_list()**

Esta operación lista el contenido de un directorio. **Es importante recalcar que al no existir subdirectorios existen solamente dos opciones de listado**

**Parámetros:**

**Directorio de destino:** Ruta absoluta al directorio a listar

**Si el parámetro es “/”**

Esta operación devolverá la lista de DAs montados, cada uno de ellos, en un directorio separado.

**Si el parámetro es “/[DA]”** (donde DA es el nombre de un dispositivo de almacenamiento

Esta operación devolverá la lista de archivos del DA enviado como parámetro.

1. Enviará al SFS la solicitud de listar los archivos de un DA. Primitiva: listarDirectorio().

**Consola**

La consola deberá soportar al menos los siguientes comandos:

**montar/desmontar/formatear([DA])**: Intentará montar/desmontar/formatear un DA previamente conectado.

**md5sum([DA], [nombre de archivo], [path en disco local]):** Utilizando el set de system calls sys\_\* deberá bajar al disco local el archivo pasado como parámetro e imprimir en pantalla su md5sum.

**Primitivas del SFS**

El SFS pondrá a disposición del VSK las siguientes operaciones, las cuales podrán ser accedidas mediante el paso de mensajes utilizando el protocolo IPC/IRC

existeArchivo(): Dado el nombre de un DA y un nombre de archivo, verificará que dicho archivo exista en el dispositivo.

infoArchivo(): Dado el nombre de un DA y un nombre de un archivo, devolverá la tabla de sectores que componen al archivo y su tamaño total.

eliminarArchivo(): Dado el nombre de un DA y un nombre de un archivo, eliminará dicho registro de la tabla de archivos y agregará los sectores que lo componían a la tabla de sectores libres usando la primitiva: liberarSectores().

crearArchivo(): Dado el nombre de un DA y un nombre de archivo, creará el registro en la tabla de archivos asignando espacio cero y con una lista de sectores vacía (sin sectores).

actualizarTamanio(): Dado el nombre de un DA, un nombre de archivo y un valor *long*, actualizará campo correspondiente al tamaño del archivo en el registro de la tabla de archivos.

crearTablaSectoresLibres(): Dado el nombre de un DA y una cantidad de sectores int, creará una lista de sectores libres desde el 0 a la cantidad N-1 para el correspondiente DA.

dosSectoresLibres(): Dado el nombre de un DA, quitará dos sectores libres de la lista y devolverá sus identificadores.

liberarSectores(): Dado el nombre de un DA y una lista de sectores, agrega dichos sectores a la tabla de sectores libres de dicho DA.

listarDirectorio(): Dado el nombre de un DA, devolverá los nombres y tamaños de todos los archivos existentes dentro de dicho DA.

formatear(): Dado el nombre de un DA y una cantidad de sectores int, vaciará la tabla de archivos y creará una nueva tabla de sectores libres invocando las primitivas correspondientes (eliminarArchivo, rearTablaSectoresLibres).

**Consola**

La consola deberá soportar al menos los siguientes comandos:

**listarDirectorio([DA])**: Imprimirá en pantalla el listado de archivos de dicho DA.

**infoArchivo([DA], [nombre de archivo]):** Imprimirá en pantalla la información correspondiente al archivo pasado como parámetro.

|  |  |
| --- | --- |
| **Modo de testeo de la entrega** | |
| * Montar una unidad de disco. Darle formato. Crear, borrar y leer archivos enviando mensajes al VSK. |

|  |  |
| --- | --- |
| **Tiempo estimado** | dos semanas |
| **Tipo de entrega** | No Obligatoria |
| **Fecha** | 28/11/2009 |

**4.5.-Quinta entrega – Entrega Final**

**Descripción**

En esta entrega se modificará las funciones dentro de la librería del servidor FTP para que en vez de utilizar la Windows API y el file system local, interactúe con la API del kernel y por ende pueda utilizar el sistema de archivos desarrollado por el alumno.

A partir de ahora la aplicación deberá enlazarse con la biblioteca en tiempo de ejecución y registrará la dirección de memoria de las funciones de la API para luego ser reemplazar a las funciones de la Windows API. La ubicación de la biblioteca será en el mismo directorio donde se encuentra el binario de la aplicación.

A continuación se describen las funciones que deberá exportar la Dll:

int **vsk\_remoteInitialize** (void \* params);

Con esta función se inicializa la biblioteca pasando la información de donde se encuentra el Kernel (dirección IP y puerto de escucha) así como algún dato más que se crea conveniente.

char \* **vsk\_remoteFList** (char \* path);

Esta función tiene como objetivo ejecutar un comando del kernel para listar el directorio path en forma remota estableciendo una conexión por cada llamado y la libera al retornar la respuesta. La información de respuesta es almacenada en un buffer temporal el cual el usuario que lo utilice será el responsable de liberar la memoria reservada para el mismo.

int **vsk\_remoteOpenFile** (const char \* fileName, int flags, int mode);

El parámetro *fileName* indica el nombre del archivo que se va a abrir según el modo de apertura indicado por la variable *flags*. El parámetro *mode* indica el derecho de acceso sobre el archivo. El valor de retorno es el descriptor de entrada/salida correspondiente al archivo abierto. Comandos ftp que lo ejecutan: put, get, delete.

Los flags son los siguientes:

|  |  |
| --- | --- |
| **VSK\_REMOTE\_READ** | Apertura del archivo para solo lectura. |
| **VSK\_REMOTE\_WRITE** | Apertura del archivo para solo escritura. |
| **VSK\_REMOTE\_DELETE** | Elimina el archivo si existe. |

Establece una nueva conexión por sockets al VSK que se mantendrá abierta mientras viva el descriptor del archivo.

int **vsk\_remoteCloseFile** (int fileDescriptor);

El parámetro *fileDescriptor* corresponde al descriptor devuelto por la llamada anterior. Retorna error si el descriptor no es válido. Además deberá cerrar la conexión asociada al descriptor establecida al abrir el archivo. Comandos que lo ejecutan: put, get, delete.

int **vsk\_remoteReadFile** (int fileDescriptor, const char \* buffer);

Esta función provoca la lectura de hasta 1024 bytes del archivo especificado por *fileDescriptor*. Los bytes son volcados a una memoria intermedia referenciada por *buffer*. La función devuelve la cantidad de bytes efectivamente leídos, cero en caso de no haber más datos para leer y un valor negativo en caso de error. La comunicación se realiza utilizando la conexión asociada al descriptor del archivo. Comandos ftp que lo ejecutan: get.

int **vsk\_remoteWriteFile** (int fileDescriptor, const char \* buffer, int size);

Esta función escribe los datos en el descriptor *fileDescriptor* que se encuentran en la memoria referenciada por *buffer,* cuya longitud en bytes se pasa en la variable *size.* La comunicación se realiza utilizando la conexión asociada al descriptor del archivo. Comandos ftp que lo ejecutan: put.

**Interface de aplicación:**

Cada una de estas funciones se comunicará remotamente con el VSK a través del protocolo IPC/IRC donde se le preparará un paquete con los datos solicitados. Las mismas son de carácter bloqueantes lo que significa que la misma no retornará hasta que haya recibido la respuesta del Kernel. El siguiente diagrama muestra la secuencia de ejecución:



**Flujo de ejecución:**

1. Al hacer uso de una rutina de la dll (por ejemplo: **vsk\_remoteOpenFile)** la dll, internamente armará un mensaje IPC/IRC con el nombre y los parámetros de la system call del VSK (**sys\_open**).
2. Dicho mensaje se lo enviará al VSK.
3. El mensaje será recibido por el VSK el cual ejecutará la rutina propiamente solicitada (**sys\_open**).
4. Generará una respuesta en IPC/IRC y la enviará.
5. La función de la dll devolverá un mensaje dependiendo del resultado enviado por el Kernel.

|  |  |
| --- | --- |
| **Modo de testeo de la entrega** | |
| * Analizar posibles problemas de concurrencia en el acceso a la API. Recordar que el servicio de transferencia corre múltiples hilos de ejecución. * Lo recomendable en esta etapa de integración es realizar el testing de stress (*Prueba diseñada para determinar la respuesta de un sistema bajo condiciones de carga*). Es importante tratar de desarrollar toda la funcionalidad con una semana (como mínimo) de anticipación a la entrega final a fin de realizar el testing de la aplicación por completo y reducir al máximo posible la cantidad de bugs. Recordar que es indispensable, para la aprobación del trabajo práctico, la entrega completa de la funcionalidad y su correcto funcionamiento. |

|  |  |
| --- | --- |
| **Tiempo estimado** | Dos semanas |
| **Tipo de entrega** | Obligatoria |
| **Fecha** | 05/12/2009 |

# 5.- Requerimientos técnicos y limitaciones

**Dynamic Link Libraries**

La creación de la Dll se realizará desde un proyecto nuevo en Visual Studio. El tipo de proyecto será para la creación biblioteca de enlace dinámico y estático. Como opción adicional no se exportarán símbolos.

*Propiedades del proyecto*

El proyecto del VS deberá ser configurado con las siguientes propiedades:

Propiedades C/C++:

* Runtime Library: Multi-Thread (/MT)
* Calling Convention: \_\_cdecl (/Gd)
* Compile As: Compile as C Code (/TC)

*Exportación de funciones*

La forma de exportación se hará utilizando el modificador *\_\_declspec(export)* por lo que no será necesario utilizar un module definition file (.def). La Dll posee una tabla de exportación con los nombres de todas las funciones que la misma exporta a otros ejecutables. Esta tabla se podrá consultar desde la herramienta DUMPBIN con la opción /EXPORT que nos brinda el Visual Studio.

*Linking*

El método de enlace que se usará en la aplicación será el enlace explicito. Por esto no será necesario enlazar la aplicación con ninguna import library (.lib). El enlace será hecho por la aplicación en run-time y para lograrlo se utilizarán las llamadas: LoadLibrary, GetProcAddress, FreeLibrary.

**Comunicaciones – Sockets**

Para la comunicación cliente/servidor se utilizarán los protocolos especificados en el trabajo práctico y se implementarán utilizando sockets (modelo Berkeley) orientados a la conexión del tipo *AF\_INET* para IPv4 (en las tres plataformas).

Para la programación de sockets en Windows, se va a utilizar la biblioteca Winsock declarada en *winsock2.h*. Se requiere linkear con la dependencia *Ws2\_32.lib*.

**Inicialización de la biblioteca Winsock**

Antes de llamar a cualquier función de la biblioteca Winsocks, se deberá inicializar dicha librería indicando la versión de Window Sockets y obteniendo los detalles de la implementación.

Finalizada la utilización de la biblioteca ó ante algún error en la inicialización de la misma, se deberá liberar los recursos y finalizar el uso de la biblioteca Winsock.

Esta inicialización se realiza **solamente una vez** por ejecución de la aplicación y no debe ser inicializada desde la Dll.

Los parámetros inicialización serán los siguientes:

**Parámetros**

*Versión solicitada*

Se utilizará la versión 2.2.

**Process Management y Multithreading**

*Windows*

Para la creación de threads en Windows se utilizará la función *\_beginthreadex* perteneciente a la *C/C++ Runtime library*. De esta manera se evitarán posibles *memory leaks.* (Para investigar: ¿cuales son las causas de este comportamiento al usar la API *CreateThread*?).

Cada thread tendrá por defecto un *stack space* de 1 megabyte.

*Solaris y Linux*

En Linux, en caso de ser requerido, solo se podrá utilizar la API de la biblioteca pthreads que forma parte del estándar POSIX.

Para la plataforma Solaris se utilizará la API de Solaris Threads para la creación y manipulación de hilos de ejecución. Se puede dar el caso en que exista funcionalidad que no esté presente en Solaris Threads y si exista en POSIX. En este caso es posible utilizar ambas para adquirir funcionalidad extra.

Cada Thread que la aplicación genere deberá tener por defecto el *stack space* en 1024 kilobytes por arriba del *PTHREAD\_STACK\_MIN*.

Para la creación de procesos, en caso de ser requerido, se utilizará el modelo *Fork-one Model*.

**Thread Synchronization**

En Windows se utilizarán Kernel Objects para la sincronización. Estos objetos serán los utilizados por las llamadas *wait functions* para coordinar la ejecución de múltiples threads al pasar de un estado señalizado a uno no señalizado.

A continuación se dará una descripción de los mecanismos que ofrece la Windows API y que estarán permitidos utilizar para la sincronización de threads en *user-mode*.

Los siguientes kernel objects podrán ser utilizados para la sincronización y queda a criterio del alumno cual utilizar según crea conveniente:

* *Events Kernel Objects*
* *Semaphore Kernel Objects*
* *Mutex*
* *Threads*
* *Waitable timers*

En Solaris y Linux también se podrán utilizar cualquier tipo de *synchronization objects*:

* *Mutex Locks*
* *Semaphores*
* *Condition variables*
* *Read-Write Locks*

**Memory Management**

*Windows*

Cada thread que la aplicación genere, creará su propio bloque de páginas adicional (su propio heap) en el espacio de dirección del proceso y al que solo él tendrá acceso. Esto lo hará para tener un manejo más eficiente de memoria y evitar el overhead generado por la sincronización entre threads. La cantidad de memoria que deberán ocupar las páginas al ser inicializadas es de 1024 Kb. No habrá restricciones en cuanto a tamaño. El tamaño máximo estará limitado por la cantidad de memoria disponible en el sistema.

Dado que el tiempo de vida de cada thread será corto, este no se ocupará de la fragmentación que se genere en el heap. Cuando el thread ya no necesite el heap este lo deberá eliminar para liberar los recursos.

En ningún momento un thread que no sea el principal utilizará la memoria del heap creado por defecto en el proceso ó heap global. Para esto se utilizarán las funciones de manejo de memoria proporcionadas por la API de Windows. No está permitido el uso de la biblioteca estándar de C para manejo de memoria (*Stdlib.h: malloc, free, etc*).

*Solaris y Linux*

No existen restricciones en cuanto al uso de memoria.

**File system**

Toda operación de I/O que involucre manejo de archivos en Windows deberá ser usando la Windows API correspondiente en su versión *ANSI* (en caso de existir *UNICODE*). No está permitido el uso de la biblioteca estándar de C para manejo archivos (*Stdlib.h: fread, fwrite, fopen, etc*).

Para Linux y Solaris no existe tal restricción.

**LDAP y OpenDS**

Como implementación de LDAP se utilizará OpenDS de SUN. Se encuentra disponible en la Virtual Machine de Solaris de la cátedra y se debe utilizar esta versión. El proyecto está escrito en Java y es multiplataforma. Requiere que el entorno tenga instalado un Java Runtime Environment. Se puede descargar gratuitamente de <http://www.opends.org/>.

Para utilizar el protocolo LDAP está permitido utilizar la API de OpenLDAP disponible en http://www.openldap.org/. Se encuentra para descargar tanto para Linux como para Solaris. También es posible la descarga con el administrador de paquetes de ambos sistemas.

La cátedra da soporte para OpenLDAP brindando un wrapper de la biblioteca original. Se puede obtener del grupo oficial de la cátedra en la sección de archivos.

**6.-Anexo – Archivo Log y Debugging**

**Manejo de Errores y Excepciones**

La aplicación mostrará por consola y generará en el correspondiente archivo Log el código de error y la descripción de dicho código obtenido del sistema tanto para las llamadas a la API de Windows como para las System Calls de Linux y Solaris. Para los errores pertinentes a la aplicación de deberá respetar las normas de logueo del trabajo práctico.

*Algunas aclaraciones para la plataforma Windows*

Dichos códigos se encuentran definidos en el header *WinError.h*. Para la obtención del *message string* a partir cierto código generado, se utilizará la función más apropiada de la API de Windows en su correspondiente versión *ANSI* (en caso de existir *UNICODE*).

**Structured Exception Handling**

*Windows, Solaris y Linux*

Queda prohibido el uso de cualquier mecanismo de exception handling ya sea para sentencias del tipo *exception handler* (\_\_try \_\_except) ó *termination handlers* (\_\_try \_\_finally) en Windows.

Si existiese una API de Windows que permita el control de errores mediante este mecanismo, generando una excepción ante un error, se deberá forzar a que retorne un código de error para ser logueado tal como se explica en la sección de **Formato del Archivo Log**.

**Formato del archivo Log**

El archivo Log deberá respetar el siguiente formato de presentación en todos los procesos que se ejecuten en cualquiera de los tres sistemas operativos.

En caso de utilizar distintos niveles detalle, el cambio entre uno u otro debe ser configurable por el usuario.

Se le recomienda al alumno registrar en este archivo los eventos más importantes de la ejecución de la aplicación, así como los valores necesarios para conocer el estado del sistema en un determinado momento. Esto es muy importante ya que en instancias finales de evaluación es probable que se haga uso de este archivo en situaciones donde la aplicación falla.

**Fecha NombreProceso [PIDproceso][ThreadID]: TipoLog: Dato**

**Descripción**

*Fecha*

Fecha del sistema. Deberá respetar el siguiente formato [HH:mm:ss.SSS].

*Nombre Proceso*

Nombre del proceso que está escribiendo en el Log.

*PID Proceso*

Process ID del proceso que está escribiendo en el Log.

*Thread ID*

ID del thread que escribe en el archivo. Opcional para el thread principal del proceso.

*TipoLog*

INFO, WARN, ERROR ó DEBUG nivel de detalle según lo que consideren apropiado.

*Data*

Descripción del evento ó cualquier información que se considere apropiada.

**7.-Anexo – Documentación**

El material de soporte para poder llevar a cabo este trabajo práctico se encuentra en su mayoría en la Web. A continuación se enumeran enlaces a páginas con la información necesaria para cada plataforma.

**MSDN de Microsoft** *http://msdn .microsoft.com*.

**Sun Microsystems Documentation** http://docs.sun.com/

* **Threads y Processes**

[http://msdn.microsoft.com/en-us/library/ms686937%28VS.85%29.aspx](http://msdn.microsoft.com/en-us/library/ms686937(VS.85).aspx)

* **Dynamic Link Libraries**

http://msdn.microsoft.com/en-us/library/ms682589(VS.85).aspx

* **Scheduling**

http://msdn.microsoft.com/en-us/library/ms685096(VS.85).aspx

* **Winsocks**

[http://msdn.microsoft.com/en-us/library/ms740673%28VS.85%29.aspx](http://msdn.microsoft.com/en-us/library/ms740673(VS.85).aspx)

* **Memory Management**

[http://msdn.microsoft.com/en-us/library/aa366779%28VS.85%29.aspx](http://msdn.microsoft.com/en-us/library/aa366779(VS.85).aspx)

* **Thread Syncronization**

[http://msdn.microsoft.com/en-us/library/ms682584%28VS.85%29.aspx](http://msdn.microsoft.com/en-us/library/ms682584(VS.85).aspx)

* **Time Functions**

<http://msdn.microsoft.com/en-us/library/ms724962(VS.85).aspx>

* **Cryptography**

<http://msdn.microsoft.com/en-us/library/aa380255(VS.85).aspx>

* **Security**

http://msdn.microsoft.com/en-us/library/cc527452.aspx

También se recomienda el siguiente material bibliográfico como soporte teórico:

* Programming Applications for Microsoft Windows, 4th edición de Jeffrey Richter.
* Microsoft Windows Internals, 4th Edición de Mark E. Russinovich y David A. Solomon.
* Solaris Internals: Solaris 10 and Open Solaris Kernel Architecture, Segunda Edición de Richard McDougall
* Understanding the Linux Kernel, 3rd Edición de Daniel P. Bovet, Marco Cesati

También pueden encontrar toda la documentación recomendada por nosotros, en nuestro grupo, accediendo a:

<http://groups.google.com.ar/group/tp-so-frba-utn/web/links-a-tutoriales?hl=es>

Información sobre cómo utilizar ***OpenLDAP SDK*** se encuentra disponible en la siguiente dirección:

<http://www.openldap.org/software/man.cgi?query=ldap>

Información sobre cómo utilizar ***OpenDS*** puede encontrarse en los siguientes enlaces:

<https://www.opends.org/wiki/page/OpenDSUserDocumentation>

Información sobre cómo utilizar ***memcached*** puede encontrase en los siguientes enlaces:

<http://www.danga.com/memcached/>

Las API clients de ***memcached*** las pueden encontrar en:

<http://code.google.com/p/memcached/wiki/Clients>

**8.-Anexo – Protocolos de comunicación**

**8.1.-Inter Process Comunications (IPCs) – Protocolo IRC/IPC standard**

Se utilizara un mensaje de protocolo interno. Estos son los campos mínimos que todo mensaje interno debe utilizar.



**Request:**

*Descriptor ID:*

Identificador de 16 bytes único descriptor en la red.

*PayloadDescriptor:*

Identificador de nro de protocolo.

*PayLoad Lenght:*

La longitud del descriptor inmediatamente seguido del header.

*Payload:*

La carga de datos que se necesite pasar. Queda libre al usuario del protocolo.

**Response:**

*Descriptor ID:*

Identificador de 16 bytes correspondiente al Request.

*PayloadDescriptor:*

Identificador de nro de protocolo.

*PayLoad Lenght:*

La longitud del descriptor inmediatamente seguido del header.

*Payload:*

La carga de datos que se necesite pasar. Queda libre al usuario del protocolo.

**8.2.-Handshake**

Para realizar los handshake entre los procesos se deberá enviar un mensaje de request del protocolo IPC/IRC con los siguientes valores:

*Descriptor ID:*

Identificador de 16 bytes único descriptor en la red.

*PayloadDescriptor:*

Nro. identificando el proceso que se está conectando, reservado para iniciar la conexión.

*PayLoad Lenght*: 0, indicando que no hay payload.

El response por su parte tendrá el siguiente formato:

*Descriptor ID:*

El identificador del request para determinar a qué mensaje pertenece.

*PayloadDescriptor:*

Nro. reservado con 2 valores posibles: ok y fail (a continuación de un response fail se deberá cerrar la conexión).

*PayLoad Lenght:* 0, indicando que no hay payload.