Universidad Tecnológica Nacional – Facultad Regional Buenos Aires
Ingeniería en Sistemas de Información
Sistemas Operativos (082027)

Trabajo Práctico

Sistemas Operativos

---

yanero

(Yet Another NEtworking RemOte filesystem)

1° cuatrimestre 2012 - v1.2-SNAPSHOT

Control de Cambios

Versión 1.1

- Página 9: Se quitó la operación ”Renombrar un archivo”, ya que la misma no debe implementarse.
- Página 13: Se modificaron los parámetros del comando mkfs.ext2
- Página 13: Se agregó la nota al pie: *“Para una buena administración de la memoria del RFS, se sugiere evitar mantener en memoria todas las estructuras administrativas y/o de datos. En caso de tener una caché interna, esta debería ser solo un subconjunto considerablemente menor a la totalidad del espacio de datos existente”.*
- Página 15: Se agregó la nota al pie: *“La lógica utilizada para definir cuanta memoria extra sera necesaria, es decir, la memoria que va a contener la key y los campos necesarios para administrar el esquema de memoria seleccionado, deberá estar argumentada teniendo como consideración principal el minimizar el uso de la memoria necesaria para esto. Esta justificación deberá ser validada con el ayudante a cargo del grupo, durante el desarrollo del TP”*
- Página 17: Se agregó el requerimiento de indicar el instante de tiempo en el que se realizó cada dump de la cache
- Página 20: Se corrigió la fecha del Tercer Checkpoint a 30/6/12
- Página 22: Se agregaron aclaraciones y limitaciones relacionas con el uso del filesystem (se sugiere releer la sección completa: “7.- Requerimientos técnicos y limitaciones > FileSystem”)

Índice

1.- Introducción

El trabajo práctico de este cuatrimestre consiste en la implementación de un sistema de archivos remoto que opere archivos con el formato ext2 sobre un sistema operativo GNU/Linux. Dicha implementación trabajará con una caché remota, la cual tendrá su propio esquema de memoria.

El objetivo es que mediante el desarrollo del mismo el alumno comprenda y se interiorice con diversos mecanismos que utiliza el sistema operativo para gestionar los medios de almacenamiento y su relación e interacción con los sistemas de archivos.

No se pretende hacer foco en el funcionamiento específico de un determinado sistema operativo sino, por el contrario, mostrar conceptos y aspectos de diseño generales.

Dada su complejidad, este trabajo fue dividido en **checkpoints,** los cuales permiten diseñar la arquitectura definitiva de forma gradual, simplificando así su desarrollo. Esta parte se encuentra detallada en el apartado **Ciclo de Desarrollo**.

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

- Procesos
  - Creación
  - IPC
    - Sockets TCP / Unix
    - Señales
  - Manejo de hilos mediante la API del sistema
    - Creación
    - Sincronización

- Memoria
  - Administración mediante particiones dinámicas
  - Administración mediante el algoritmo Buddy System

- Sistema de Archivos
  - Implementación de ext2

- Diseño de SO
  - Arquitecturas basadas en capas físicas y lógicas
  - Protocolos

2.- Objetivos del Trabajo Práctico

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

- Adquiera los conocimientos prácticos del uso y aplicación de un conjunto de servicios que ofrecen los sistemas operativos modernos.
- Domine los problemas específicos de este tipo de implementaciones.
- Evalúe las ventajas y desventajas de la utilización de distintas soluciones para un mismo problema.
- Afronte problemas de diseño en los componentes propios de un sistema operativo.
- Entienda la importancia de una norma o protocolo estándar en la comunicación entre procesos y diferentes plataformas.
- Se entrene en el trabajo en equipo, manejo de las problemáticas de un grupo y las responsabilidades que esto implica.

Esta consigna de trabajo práctico ***no es una especificación completa***, sino que brinda los lineamientos principales de los objetivos a cumplir. Cualquier detalle que en la consigna para el trabajo práctico no esté especificado y necesite ser definido para el desarrollo del mismo, deberá ser consultado y, si no resulta ser un error de enunciado, *el grupo deberá tomar sus propias decisiones de implementación, asegurándose que estén avaladas por el ayudante a cargo (se sugiere documentarlas)*.

El objetivo de proponer una especificación más abierta es que los grupos aprendan a analizar diferentes alternativas para resolver un mismo problema y que, al mismo tiempo, evalúen los costos y beneficios de implementar cada una de estas alternativas.

3.- Características del Sistema

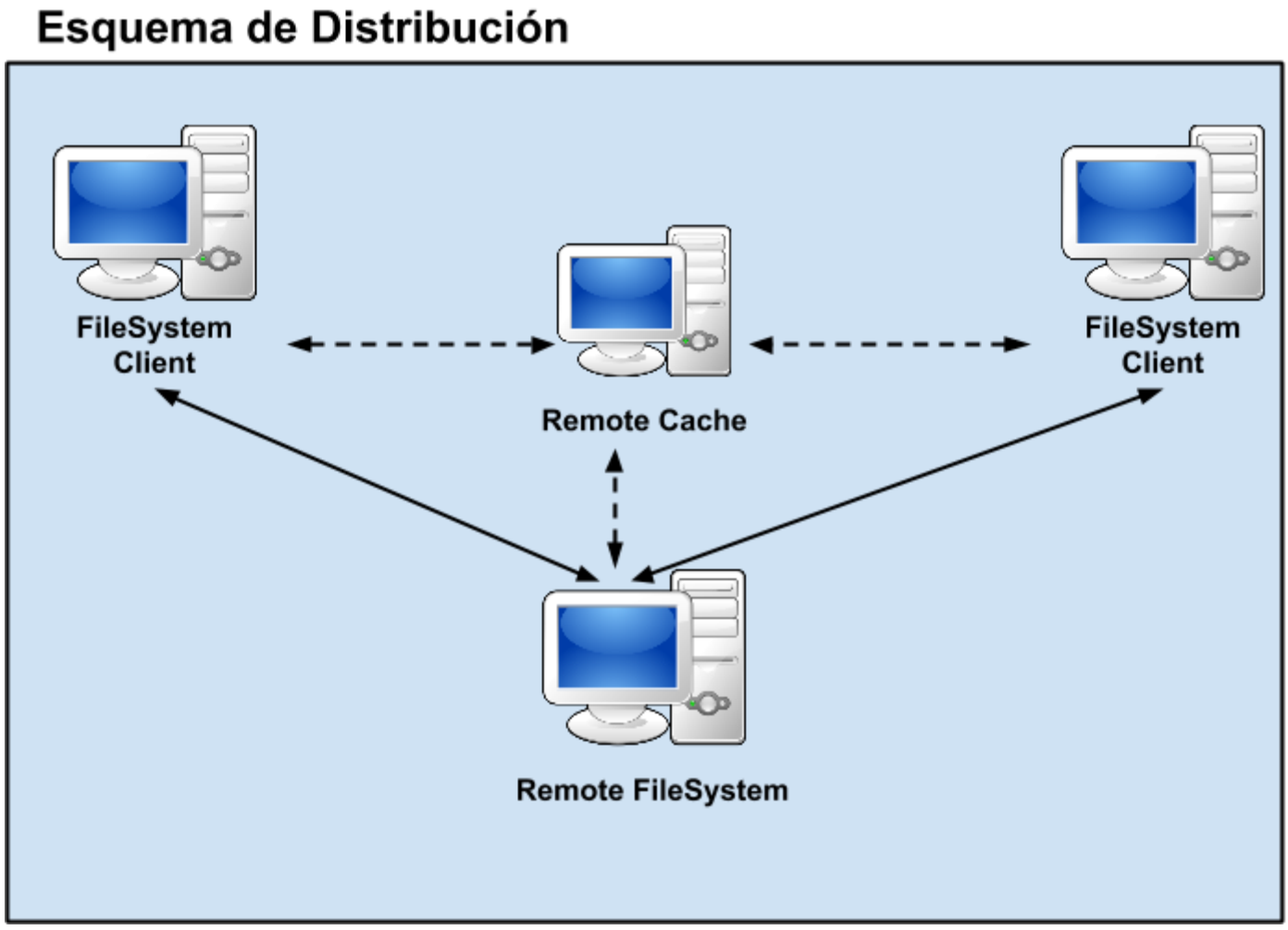
Definición

El sistema a implementar tendrá como objetivo permitir montar un único sistema de archivos instalado en un servidor, desde diferentes máquinas clientes en una red local. El formato del sistema de archivos será ext2.

El mismo contará con el apoyo de una caché, que estará a disposición tanto de los clientes como del servidor, en pos de favorecer el desempeño de las distintas operaciones que el sistema proveerá.

Diagrama de componentes

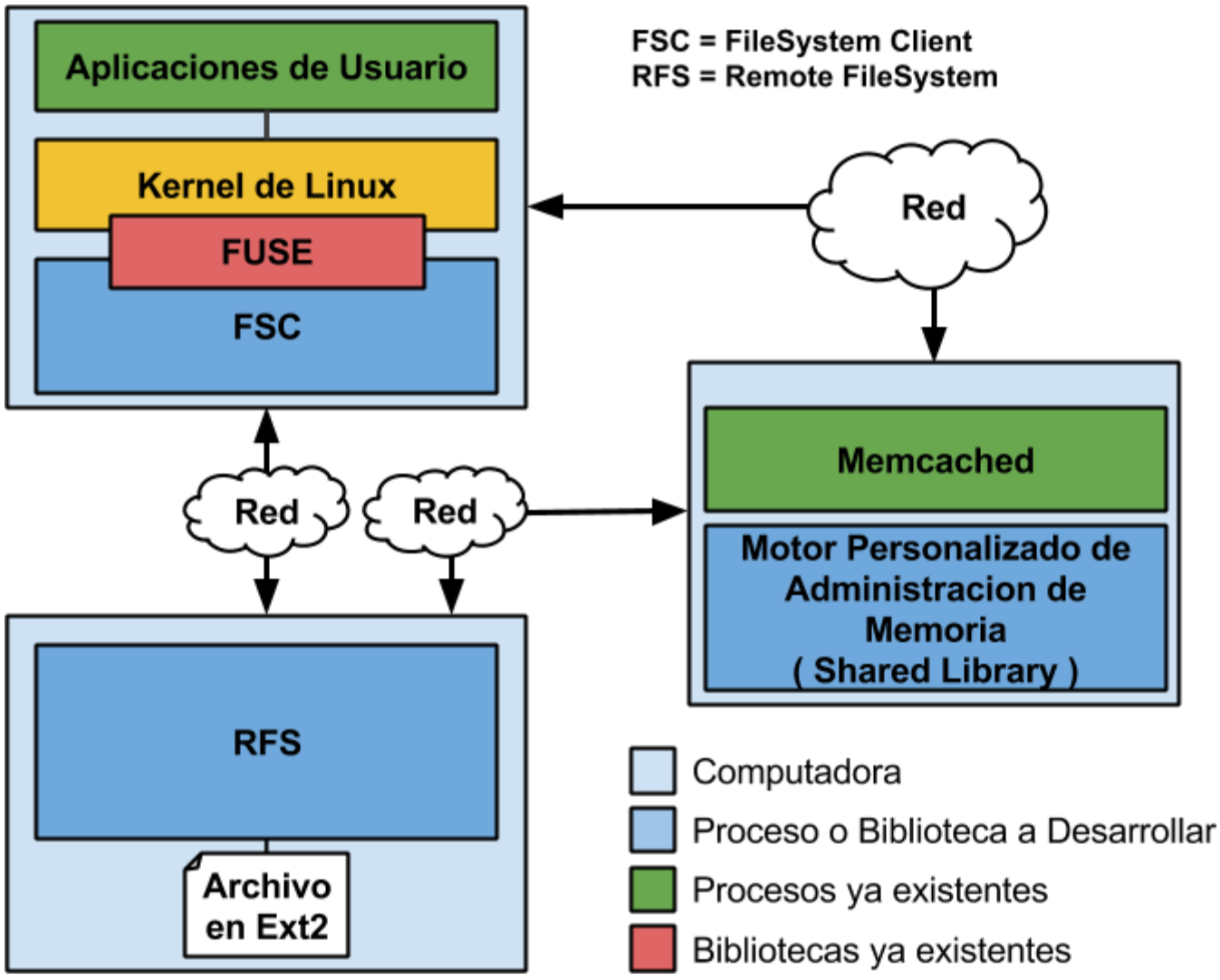
El sistema estará conformado por tres componentes, bajo el siguiente esquema:



En el esquema presentado, el Remote FileSystem expone las funciones/operaciones que se pueden realizar sobre el archivo particionado en ext2 a dos FileSystem Client (sin embargo, podrían ser más clientes). Estos clientes harán de interfaz entre el sistema operativo de la computadora local y el Remote FileSytem, utilizando FUSE.

A la par de esto, los resultados de las operaciones provistas tanto por el FSC como por el RFS serán cacheadas en el servidor Memcached.

Esquema por Capas



FileSystem Client (FSC)

Este componente será un **Proceso** que interactuará con el kernel del sistema operativo anfitrión para brindar los servicios necesarios para operar remotamente con el sistema de archivos ext2.

Ésto se realizará utilizando la biblioteca FUSE, la cual permitirá a este componente actuar como un proxy (puente) entre el Remote FileSystem y el sistema operativo host.

Remote FileSystem (RFS)

Este componente será un **Proceso**, y estará encargado de leer, interpretar y modificar el sistema de archivos ext2 como consecuencia de las operaciones que se expondrán a los clientes.

Archivo de Volumen en ext2

Este componente es simplemente un **archivo** regular que contiene los datos de un disco formateado con el sistema de archivos ext2, lo que le permite simular ser un disco real sobre el cual va a interactuar el Remote FileSystem.

Por cada archivo existirá sólo una instancia del proceso que lo interprete y exponga sus operaciones.

Remote Cache (RC)

Este componente será un ***Proceso*** que brindará el servicio de caché en modo remoto, tanto al FSC como al RFS.

El proceso en sí mismo será creado por la herramienta Memcached, provista por la cátedra. **La labor del grupo consistirá en implementar una Shared Library** (biblioteca compartida) que será el motor personalizado de almacenamiento de memoria. De esta manera, el proceso Memcached utilizará la shared library desarrollada por el grupo para almacenar y recuperar valores.

4.- Aspectos funcionales de los procesos

FileSystem Client (FSC)

Este proceso actuará como un puente que comunicará al kernel del sistema operativo host con el Remote FileSystem. Implementará FUSE para comunicarse con el kernel y exponer las funcionalidades, y utilizará sockets mediante las cuales se conectará con el Remote FileSystem y enviará las operaciones.

Al ejecutar el proceso, recibirá por parámetro, *como mínimo*, el path (ruta) del directorio donde se montará el sistema de archivos ext2. A partir de aquí, el proceso quedará a la espera de las solicitudes de operaciones por parte de las aplicaciones de usuario, de manera concurrente.

Deberá implementar las siguientes operaciones:

- Crear un archivo
- Abrir un archivo
- Leer un archivo
- Escribir un archivo
- Borrar un archivo
- Truncar un archivo
- Cerrar un archivo
- Crear un directorio
- Leer un directorio
- Borrar un directorio
- Obtener los atributos de un archivo/directorio

Remote File System (RFS)

Este proceso es el encargado de leer una partición en formato ext2 y de exponer las operaciones que se pueden realizar sobre el mismo a los distintos clientes que se le conecten.

Al ejecutar el proceso, recibirá por archivo de configuración el path (ruta) donde se encuentra el archivo formateado en ext2. A partir de aquí, se pondrá a la escucha de potenciales conexiones de clientes, que solicitarán algunas de las operaciones antes mencionadas.

Pondrá a disposición un servicio (en forma de mensaje) por cada operación que el FSC pueda solicitar.

Remote Cache (RC)

Este proceso tendrá el rol de una cache remota, la cual será accedída por múltiples clientes de forma concurrente.

Memcached se encargará de interpretar el protocolo de comunicación y delegar las operaciones en varios hilos. El proceso será provisto por la cátedra y no necesitará ser modificado funcionalmente por el grupo.

La parte a desarrollar será el motor, que Memcached utilizará, encargado de administrar la memoria para satisfacer los pedidos de almacenamiento de información provenientes de los clientes, y su posterior lectura. Para lograr esto, se deberá desarrollar una implementación de la interfaz proporcionada por Memcached, compilándola como una shared library.

Al momento del inicio de Memcached, se colocará como parámetro el path a la shared library generada y Memcached se encargará de invocar a la funcionalidades dependiendo de los comandos que le lleguen.

El objetivo de esta biblioteca a desarrollar es que pueda brindar las siguientes operaciones a Memcached:

- Guardar un valor
- Leer un valor
- Remover un valor
- Limpiar toda la cache

Configuración

En la especificación de cada proceso se mencionará un archivo de configuración en el cual se detallarán campos obligatorios para lograr las funcionalidades especificadas. El grupo, sin embargo, puede sentirse libre de agregar los campos extra que necesite y que sean requeridos por su implementación.

5.- Aspectos Técnicos de los Procesos

FileSystem Client (FSC)

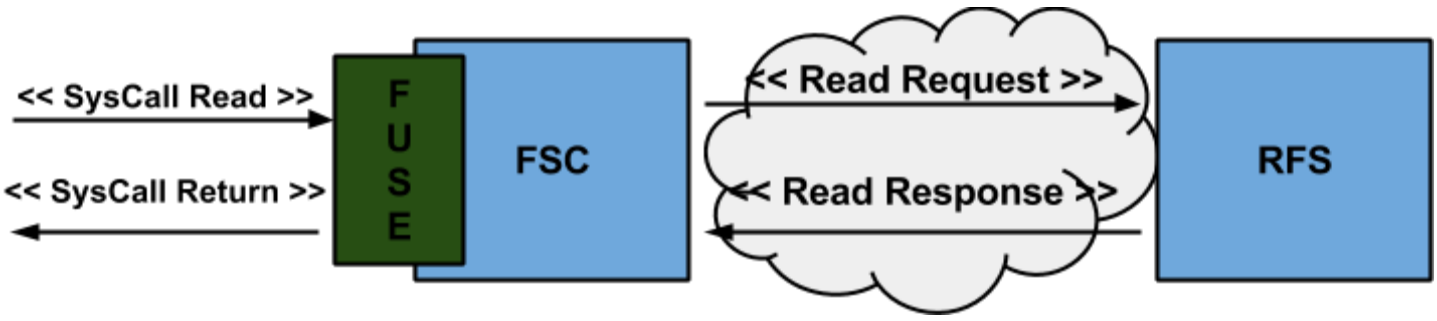
Para exponer funcionalidades de un File System al Kernel de Linux los alumnos deberán utilizar la biblioteca FUSE (ver **Anexo C**).

Las operaciones que tienen que ser implementadas en FUSE son las correspondientes a las enumeradas previamente, a saber:

- *create* /** Create and open a file */
- *open* /** File open operation */
- *read* /** Read data from an open file */
- *write* /** Write data to an open file */
- *release* /** Release an open file **/
- *truncate* o *ftruncate* /** Changes the size of file **/
- *unlink* /** Remove a file */
- *mkdir* /** Create a directory */
- *readdir* /** Read directory */
- *rmdir* /** Remove a directory */
- *fgetattr* o *getattr* /** Get file attributes. */

El proceso debe ser capaz de interactuar con el RFS. Para ello el grupo deberá usar, **y podrá extender**, el protocolo NIPC[1] especificado en el **Anexo B** para comunicar ambos procesos. Por cada una de las operaciones especificadas anteriormente el RFS proveerá un servicio asociado. Esto quiere decir que por cada operación (Véase *create, open, read, etc* …) existirá un tipo de paquete de comunicación entre ambos procesos el cual indicará el tipo de operación y los argumentos necesarios.

Ej:



En este caso se ejemplifica cómo al kernel le llega la SysCall (llamada al sistema) Read, y

[1] Network Inter Process Comunication

esta es delegada a FUSE. Dentro del FSC, FUSE invocará a la implementación de Read realizada por el grupo. Para procesar este pedido se generará un paquete, indicado en el diagrama como “<< Read Request >>”, el cual deberá contener los parámetros necesarios para que el RFS pueda procesar la operación. Por ejemplo:

- El path (ruta) al archivo
- El offset a partir de donde queremos leer
- El tamaño de lo que queremos leer

El RFS responderá con un paquete, indicado como “<< Read Response >>” en el diagrama, el cual devolverá la cantidad de información solicitada del archivo especificado en el offset (desplazamiento) indicado; o un mensaje de error en caso de que no se pueda llevar a cabo la operación.

De manera similar al ejemplo anterior, se deberán implementar los paquetes para poder satisfacer todas las operaciones indicadas.

En caso de que el RFS no esté disponible, el proceso FSC finalizará su ejecución indicando dicha situación.

Concurrencia

Como todo sistema de archivos, recibirá peticiones concurrentemente. Es decir, una aplicación “A” podría estar abriendo un archivo, mientras una aplicación “B” podría estar leyendo otro. Incluso un mismo archivo podría estar siendo accedido al mismo tiempo desde varias aplicaciones diferentes. Durante la evaluación, FUSE funcionará en modo multithread (multihilo), dando la posibilidad de que se realicen operaciones en simultáneo.

Cache

Debido a que los accesos al RFS implican tiempo teórico de acceso a disco, algunas de las operaciones deberán ser primero consultadas al RC, y en caso de que éste no contenga la información deseada entonces se accederá al RFS. Debido a que el RC será una instancia de Memcached en sí, todos los valores serán guardados mediante una key (clave), la cual estará en formato string.

Por otro lado, en caso de que la operación no se encuentre en cache, al recibir la respuesta del RFS y comprobar que la misma no indique error, se deberá enviar la respuesta obtenida, en el formato más conveniente para el grupo, al RC para que ésta quede almacenada.

La operaciones que el FileSystem Client deberá cachear son:

- *readdir*
- *fgetattr* o *getattr*

Tener en cuenta que otras operaciones, como el `truncate` por ejemplo, afectarán información previamente almacenada en la caché. Se deberán considerar dichas situaciones tomando alguna decisión en pos de la coherencia de los datos que el FSC provea a sus clientes.

La comunicación con Memcached será a través de la biblioteca libMemcached, la cual estará incluida en las VMs provistas por la cátedra.

Remote File System (RFS)

Este proceso interactuará directamente con el sistema de archivos ext2 manipulando su formato para brindar las operaciones descriptas previamente:

- Crear un archivo
- Abrir un archivo
- Leer un archivo
- Escribir un archivo
- Borrar un archivo
- Truncar un archivo
- Cerrar un archivo
- Crear un directorio
- Leer un directorio
- Borrar un directorio
- Obtener los atributos de un archivo/directorio

El medio sobre el cual este proceso realizará la lectura y/o escritura no será una partición de un disco rígido real. A fines prácticos, éste trabajará sobre un archivo regular formateado bajo ext2. Este archivo puede ser creado mediante el comando **`mkfs.ext2`**.

Este comando es capaz de crear un archivo formateado como ext2, como si se tratara de una partición de un disco, especificándole parámetros como tamaño de sector, cantidad de sectores por cluster, etc. Por ejemplo:

```
$touch ext2.disk  # Crear el archivo vacío

$mkfs.ext2 -F -O none,sparse_super -b 1024 ext2.disk 30000  # Formatearlo
```

Para probar la integridad del sistema de archivos en cualquier momento, se puede utilizar el comando `mount`, para montar el archivo como si se tratase de unidad de disco físico, utilizando el filesystem *nativo* de ext2 provisto por el sistema operativo anfitrión:

```
$ mount -o loop <path de nuestro archivo> <punto de montaje>
```

El método para operar (léase: escribir/leer[2]) sobre el archivo, deberá ser elegido (previa investigación y evaluación) de entre una de las siguientes dos alternativas:

- Unlocked Stream Operation[3]
- Mapping File Into Memory

Si bien solo debe ser elegida e implementada una opción, el alumnado debe ser consciente de ambas ya que podrán ser preguntadas en el coloquio, como así también el criterio por el cual se ha elegido la alternativa implementada. Junto con la opción que elija, el grupo deberá investigar y utilizar una de la siguientes funciones, según la opción elegida previamente:

- *posix_fadvise* para Unlocked Stream Operation
- *posix_madvise* para Mapping File Into Memory

La utilidad de la misma, y la forma en la que fue empleada, será evaluada durante el coloquio.

Concurrencia

A este proceso se le pueden conectar múltiples clientes y a su vez cada uno de estos pueden realizar múltiples operaciones concurrentemente. Para las regiones críticas, el alumno deberá optar por alguna de las siguientes herramientas:

- **Mutex** (pthread_mutex_lock, pthread_mutex_unlock, etc …)
- **Read - Write Locks** (pthread_rwlock_rdlock, pthread_rwlock_wrlock, etc …)

Como consideración al emplear alguna de las opciones de sincronización, esta *prohibido* el uso de locks globales ya que esto limitaría a que solo una operación pueda realizarse a la vez en todo el filesystem. Es por ello que deberán diseñar un esquema tal, que permita maximizar la cantidad de operaciones concurrentes que se puedan realizar sobre el filesystem.

Las conexiones con los diferentes FSC deberán ser multiplexadas, y ante la llegada de una nueva operación se le deberá asignar un thread (hilo) que la ejecute. La forma en la que se asignen los threads deberá ser una de las siguiente opciones:

- Crear un pool de threads
- Crear los threads y finalizarlos a medida que sea necesario

En cualquiera de los casos, la cantidad máxima de threads que se encuentran en ejecución, además del thread principal, estará indicado en un archivo de configuración.

Cache y Retardo

[2] Para una buena administración de la memoria del RFS, se sugiere evitar mantener en memoria *todas* las estructuras administrativas y/o de datos. En caso de tener una caché interna, esta debería ser solo un subconjunto considerablemente menor a la totalidad del espacio de datos existente.

[3] Tipear "`man unlocked_stdio`" en una terminal para más información sobre dicho mecanismo

El proceso RFS también accederá al RC, para consultar (y eventualmente almacenar) los resultados de la siguientes operaciones:

- *read*
- *write*

Por otro lado, deberá poder setearse por archivo de configuración el **retardo**, tanto en milisegundos como en microsegundos (ej: 10ms, ó 10000us), que toda operación debería tener para simular el acceso a disco. En caso de setearse en 0, no se realizará ningún retardo. Este valor **deberá poder actualizarse en tiempo de ejecución**. Para esto se pueden emplear diversas alternativas (utilizar un thread dedicado, la herramienta inotify de Linux, etc).

Remote Cache (RC)

Será el proceso provisto por la cátedra, el cual utilizará la shared library desarrollada por el grupo.

Mientras el proceso Memcached se encarga de administrar los threads, leer los parámetros de inicio, manejar el envío y recepción de paquetes e interpretar el protocolo de comunicación, la biblioteca desarrollada por el grupo va a ser la encargada de administrar la memoria. Como la biblioteca esta atada al funcionamiento de Memcached, todos los elementos que se almacenen estarán asociados a una key (clave), que deberá ser almacenada junto con el valor.

Las operaciones que el motor implementará son:

- `get` `/* Devolver un valor */`
- `store` `/* Almacenar un valor */`
- `remove` `/* Eliminar un valor */`
- `flush` `/* Vaciar toda la cache */`

Se almacenarán tanto valores nuevos como valores ya existentes. Por lo tanto, se deberá considerar la posibilidad de que se sobreescriba una key cuyo valor cambie en contenido y tamaño.

Administración de Memoria

Uno de los requerimientos obligatorios que va a tener la biblioteca será que **una vez inicializada la biblioteca ya no se podrá alocar mas memoria dinámica**. Por lo tanto, toda la memoria que vaya a ser necesaria, tanto para guardar la key, el valor y las estructuras auxiliares, deberán ser pre-alocadas dinámicamente en el inicio.[4]

[4] La lógica utilizada para definir cuanta memoria extra sera necesaria, es decir, la memoria que va a contener la key y los campos necesarios para administrar el esquema de memoria seleccionado, deberá estar argumentada teniendo como consideración principal el *minimizar* el uso de la memoria necesaria para esto.

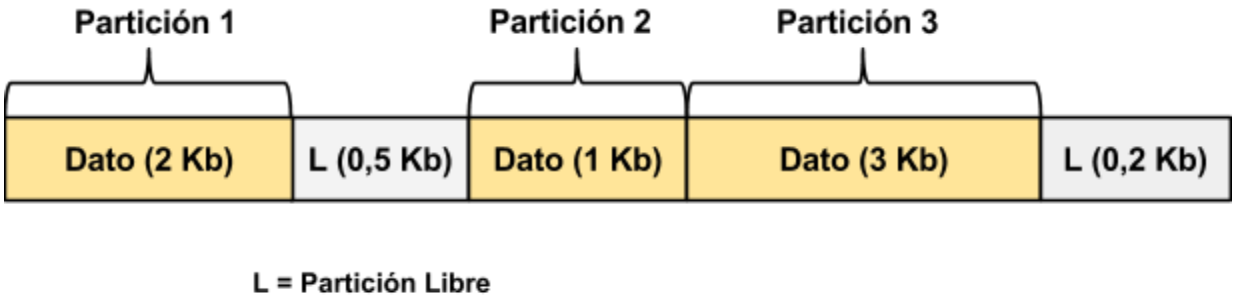
Para verificar esto, la biblioteca deberá utilizar la función `mtrace()` para registrar el uso de memoria dinámica. Dicho registro se guardará en un archivo que luego podrá ser interpretado con el comando `mtrace`.

Adicionalmente para evitar que el sistema operativo pueda llegar a penalizar la performance al tratar de swappear[5] las paginas de memoria que sean alocadas, será necesario emplear la función `mlock()`.

Se implementarán dos esquemas de Administración de Memoria: Particiones dinámicas con compactación, y Buddy System (descriptos más adelante). Se elegirá por archivo de configuración cual estará activo al iniciar la caché. Para ambos, se definirá por parámetro de Memcached el tamaño mínimo de partición y un tamaño máximo (que será el de toda la memoria).

Esquema 1 - Particiones dinámicas con Compactación[6]

En este esquema, se alocará una porción de memoria por cada valor almacenado, del tamaño exacto de dicho valor. De esta manera, la cantidad de particiones y su tamaño es variable. Por ejemplo:



En dicho ejemplo, en el caso de almacenar un nuevo valor de 0,2 Kb en el espacio de la primera partición libre, se tendría una nueva “partición 4” de 0,2 Kb, y al lado una nueva partición libre de 0,3 Kb.

Procedimiento para almacenamiento de datos

1. Se buscará una partición libre que tenga suficiente memoria continua como para contener el valor. En caso de no encontrarla, se pasará al paso siguiente (si corresponde[7], en caso contrario se pasará al paso 3 directamente).
2. Se compactará la memoria y se realizará una nueva búsqueda. En caso de no encontrarla, se pasará al paso siguiente.

Esta justificación deberá ser validada con el ayudante a cargo del grupo, durante el desarrollo del TP.

[5] Swappear: enviar a un almacenamiento secundario una porción de memoria. El concepto será visto en la clase teórica de Memoria Virtual.

[6] Referencias bibligráficas: sección 7.2, cap. 7, Stallings 6° ed.; sección 8.3.2/3, cap. 8, Silberschatz 7° ed.

[7] Se deberá poder configurar la frecuencia de compactación (en la unidad “cantidad de búsquedas fallidas”). El valor -1 indicará compactar solamente cuando se hayan eliminado todas las particiones.

3. Se procederá a eliminar una partición de datos, y luego se volverá al paso 2 o al 3, según corresponda.

Algoritmos para elección de partición libre y elección de victima

Para seleccionar una partición libre, se deberá optar por implementar alguno de los siguientes pares de algoritmos:

- Next Fit (siguiente ajuste) y Worst fit (peor ajuste).
- First Fit (primer ajuste) y Best Fit (mejor ajuste).

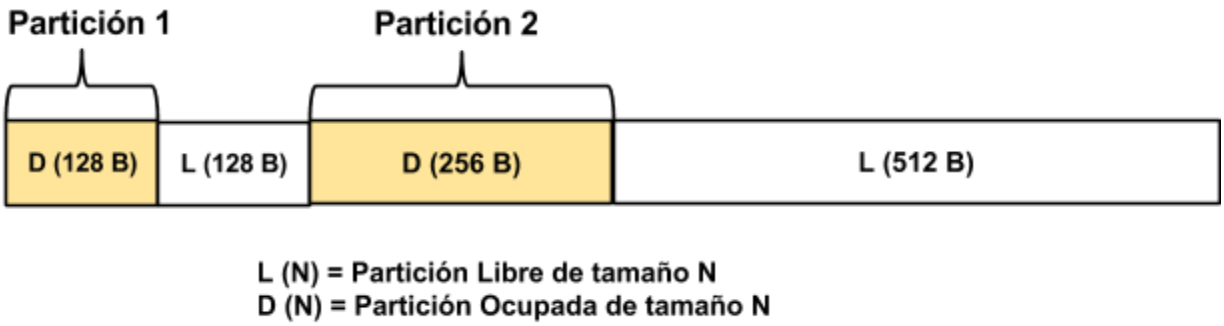
En el caso de tener que eliminar una partición, los algoritmos a implementar serán:

- FIFO (First In First Out) y LRU (Last Recently Used).

En ambos casos, el algoritmo a utilizar se definirá por archivo de configuración.

Esquema 2 - Buddy System[8]

En este esquema, se alocará una partición de memoria por cada valor almacenado, del tamaño *potencia de 2* que sea más cercano a dicho valor. Por ejemplo:



En dicho ejemplo, en caso de almacenar un nuevo valor de 63 B en el espacio de la primer partición libre, generaría una nueva “partición 3” de 64 B, y al lado quedaría una nueva partición libre de 64 B.

El procedimiento de almacenamiento de datos será similar al de las particiones dinámicas, con la salvedad que la compactación se realizará de acuerdo a las reglas del propio algoritmo Buddy Sytem. FIFO y LRU serán los algoritmos a implementar para la elección de víctima en un reemplazo (al igual que en el algoritmo previo, modificable por archivo de configuración)

Concurrencia

Al igual que el RFS, Memcached recibirá multiples pedidos concurrentes, aunque la gestión de threads no será responsabilidad del grupo. Sin embargo deberán ser tomadas las medidas

[8]Referencias bibligráficas: sección 7.2, cap. 7, Stallings 6° ed.;.

necesarias para evitar accesos concurrentes que puedan corromper la información dentro de la cache. Al igual que con el RFS, el grupo deberá optar por utilizar ***Mutex*** o ***Read - Write Locks***.

Dump de la Cache

Será requerimiento del motor de administración de memoria que éste pueda depositar en un archivo del estado actual de la memoria en la cache según el esquema seleccionado. Para solicitar dicho dump, se enviará una señal SIGUSR1 que deberá ser manejada e inicializada en la Shared Library.

No se pretende ver el contenido de la información almacenada, sino las particiones asignadas/libres, indicando su dirección de comienzo y fin, su tamaño en bytes, tiempos según el algoritmo y la clave asociada en caso de tener un valor.

Ejemplo:

Dump: 14/07/2012 10:11:12
Partición 1: 0x000 - 0x3FF. [X] Size: 1024b LRU:<VALOR> Key: "dir-/home/guest"
Partición 2: 0x400 - 0x409. [L] Size: 9b
Partición 3: 0x40A - 0x40B. [L] Size: 1b

6.- Ciclo de Desarrollo

Es una planificación propuesta por la Cátedra, mediante la cual se establecen los plazos y objetivos ideales para los cuales los grupos deberían desarrollar el Trabajo Practico. Ésto quiere decir, que se establecen una serie de puntos de control -***checkpoints***- los cuales tienen un conjunto de objetivos a desarrollar y un plazo en el que deberían estar finalizados. No se debe confundir un checkpoint con "entrega", ya que esta última implica de manera obligatoria tener todos los objetivos cumplidos para la fecha fijada.

Los checkpoints, son un punto de referencia para el alumnado y para que este comprenda las dificultades y el tiempo que demandarían cada uno de los objetivos. A su vez, el puntual cumplimiento de los checkpoints le permitiría al grupo un desarrollo más distribuido y progresivo.

Los checkpoints serán seguidos por los ayudantes asignados a cada grupo, a excepción de los que son presenciales. Esto se encuentra explicado en mayor detalle en las **Normas del Trabajo Practico**.

El desarrollo de cada uno de los tres procesos está planificado de manera paralela, por lo que los tres comenzarán su desarrollo en el primer checkpoint y finalizarán antes de la entrega final o coincidiendo con esta. Esto permite distribuir el desarrollo de manera mas uniforme y balanceada entre todos los integrantes del grupo. Cada uno de los tres procesos tiene una carga, la cual implica cuantos desarrolladores debería, aproximadamente, tener trabajando sobre este a lo largo de todos los checkpoints:

Proceso Cliente del FileSystem: 0.5
Proceso Servidor del FileSystem: 2
Proceso Cache: 1.5

Este peso no indica que las asignaciones sean fijas, sino que puede haber rotaciones dentro del grupo, lo cual seria una situación ideal.

Por ultimo cabe aclarar que pueden existir tareas fuera de lo ponderado anteriormente y que queda a discreción del grupo su manejo y planificación. Como puede ser el caso de bibliotecas genéricas como: wrappers de sockets, funciones de serialización, etc, que afectan a los tres procesos.

6.1.- Primer Checkpoint

En esta primera etapa comienza el desarrollo de los tres procesos, se considera que al menos una semana y media sería utilizada por el grupo para investigar todos los temas relevantes al

sistema, definir tareas, definir estructuras y tener un diseño inicial del sistema. El resto del tiempo debería ser utilizado en implementar las estructuras básicas y primeras funcionalidades.

Objetivos:

- **Investigar**
  - ext2.
  - FUSE.
  - Memcached.
  - Algoritmos de Memoria.

- **Proceso Remote FileSystem**
  - Leer superbloque
  - Leer Group Descriptor
  - Leer Grupos de Bloques
  - Leer bitmaps
  - Leer tabla de inodos
  - Leer inodos
  - Leer indireccionamientos
  - Buscar bloques libres
  - Buscar inodo libre

- **Proceso FileSystem Client**
  - Implementar interfaz de FUSE
  - Definición de paquetes entre cliente y servidor
  - Implementar paquetes para la operación getattr
  - Implementar paquetes para la operación readdir
  - Implementar paquetes para la operación open
  - Implementar paquetes para la operación release
  - Implementar paquetes para la operación read
  - Implementar paquetes para la operación write
  - Implementar paquetes para la operación create
  - Implementar paquetes para la operación unlink
  - Implementar paquetes para la operación truncate
  - Implementar paquetes para la operación rmdir
  - Implementar paquetes para la operación mkdir

- **Proceso Remote Cache**
  - Implementar la interfaz de memcached con funciones dummy
  - Alocación de toda la cache
  - Comienzo de implementación de uno de los esquema de memoria

Tiempo Estimado	4 semanas
Fecha de Finalización	19/5/2012

6.2.-Segundo Checkpoint [Presencial]

Objetivos:

- **Proceso Remote FileSystem**
  - Listar un directorio
  - Leer un archivo
  - Sincronización
  - Delegar operaciones en threads
  - Multiplexar conexiones

- **Proceso FileSystem Client**
  - Implementar mecanismo de recepción e interpretación de paquetes para el RFS
  - Acceso a la cache y almacenamiento

- **Proceso Remote Cache**
  - Finalizar la implementación de uno de los esquemas de memoria
  - Soporte para concurrencia en ese esquema.

Tiempo Estimado	3 semanas
Fecha de Finalización	9/6/2012

6.3.-Tercer Checkpoint

Objetivos:

- **Proceso Remote FileSystem**
  - Escribir/Modificar un archivo
  - Truncar un archivo

- **Proceso FileSystem Client**
  - Testing

- **Proceso Remote Cache**

- Comienzo de implementación del otro de lo esquema de memoria
- Soporte para concurrencia en ese esquema.

Tiempo Estimado	3 semanas
Fecha de Finalización	30/6/2012

6.4.- Entrega Final

Objetivos:

- **Proceso Remote FileSystem**
  - Crear un directorio
  - Remover un directorio
  - Crear un archivo
  - Remover un archivo
  - Testing

- **Remote Cache**
  - Finalizar la implementación de uno de lo esquemas de memoria
  - Dump de la Cache

Tiempo Estimado	2 semanas
Fecha de Finalización	14/7/2012

7.- Requerimientos técnicos y limitaciones

Compilación

Se deberá utilizar el compilador gcc, bajo cualquiera de los estándares que soporta: C90, C99 o C11.

Comunicaciones – Sockets

Para la comunicación cliente/servidor se utilizarán los protocolos especificados en el trabajo práctico y se implementarán utilizando sockets orientados a la conexión del tipo AF_INET para TCP/IPv4.

Cuando sea necesario utilizar una función para multiplexar I/O, el grupo deberá optar por utilizar alguna/s de las siguientes opciones:

- `select()`
- `poll()`
- `epoll()`

Siendo requerido, para cualquiera de los casos, el conocer las diferencias técnicas o implementaciones de cada una de ellas.

Tipos de Datos y Arquitecturas

Para tratar de asegurar una mayor portabilidad entre arquitecturas, el alumnado deberá evitar el uso del tipo **int**, donde sea necesario tener un tamaño fijo de datos, ya que este mismo cambia su tamaño dependiendo de la arquitectura. En reemplazo se deberá investigar un tipo de dato equivalente pero que asegure un tamaño sin importar la arquitectura que se maneje.

Memoria

Para hacer un seguimiento del uso de la memoria que utilice la cache, **será obligatorio utilizar la función mtrace,** la cual debe ser invocada desde la shared library al comienzo de esta.

FileSystem

- Se deberá implementar la interpretación de la revisión 0 (cero) de la especificación de EXT2, soportando adicionalmente la característica “Sparse Superblock”. Para esto, se

creará un volumen con la revisión 1, quitando todas las características nuevas, a excepción de la mencionada inicialmente.

- Para la característica “Sparse Superblock” no será necesario mantener las copias del Superblock o del Block Group Descriptor Table actualizadas. Sino que solo se deberán trabajar con las de correpondiente al Block Group 0.
- Para simplificar la carga, al momento de tener que asignar bloques libres a un inodo no será necesario que se aplique criterio alguno de localidad de bloques en los grupos de bloques.
- No se crearán archivos cuyas rutas relativas al punto de montaje sean mayores a 40 caracteres.
- Solo se manejarán los tipos de archivo “directory” y “regular file”.
- No será necesario dar soporte a las fechas de creación o acceso.
- No será necesario dar soporte a los permisos o dueños del archivo.
- No se trabajará con archivos más grandes de 1 GB.
- No se trabajará con filesystems mayores a 2 GB.
- El campo “tipo de archivo” dentro de la “entrada de directorio” debe ser ignorado.
- Las operaciones de read y write nunca van a trabajar con más de 32 KB de datos.
- El FileSystem podrá ser creado con bloques de 1Kb, 2Kb o 4Kb en el momento de las pruebas.

Process Managment y Multithreading

En caso de ser requerido, solo se podrá utilizar la API de la biblioteca pthreads que forma parte del estándar POSIX (NPTL – Native POSIX Thread Library).

Manejo de Errores

Queda prohibido el uso de cualquier mecanismo de exception handling (__try __except) o funciones/bibliotecas de C++.

8.- Anexos

Anexo A :: Archivo Log y Debugging

Registro de Errores

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 para la ejecución fallida de cualquier función del sistema invocada. Para los errores pertinentes a la aplicación se deberá respetar las normas de logueo del trabajo práctico.

Formato del archivo Log

Todos los archivos de Log deberán respetar un mismo formato de presentación. 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 o se requiera hacer un seguimiento de la ejecución, por lo que este tiene que ser legible.
El formato a respetar es, logueando **cada evento en una única línea**, el siguiente:

> *[TIPOLOG] TIMESTAMP NOMBREPROCESO/(PID:TID): MENSAJE*
>
> Ejemplo: `[DEBUG] 15:05:10:1123 PROC1/(1555:15): Creo el socket`

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.
- ***Tipo de Log:*** INFO, WARN, ERROR ó DEBUG nivel de detalle según lo que consideren apropiado.
- ***Data:*** Descripción del evento ó cualquier información que se considere apropiada.

Eventos de logueo obligatorio

Algunos eventos deberán ser logueados obligatoriamente con cierta información de contexto, con opción para deshabilitarlos por archivo de configuración, a saber:

- **Proceso Remote FileSystem**
    - Nuevos Clientes (INFO)
    - Caída de Clientes (INFO)
    - Operación solicitada, indicando:
        - Nombre del Archivo (DEBUG)
        - Tipo de operación (DEBUG)
        - En caso de lectura o escritura, el size y el offset. (DEBUG)

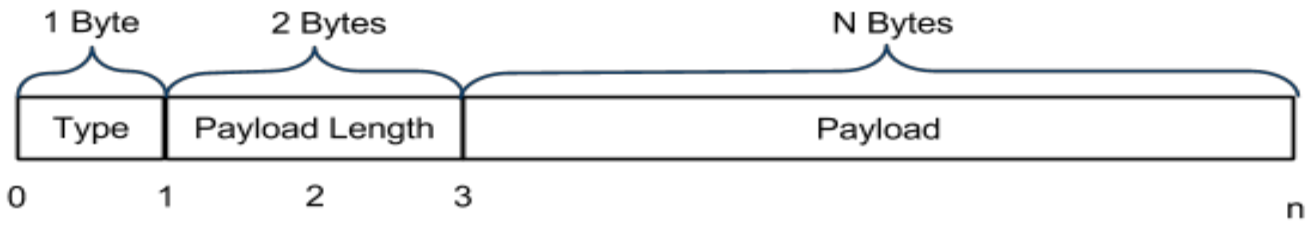
- **Proceso FileSystem Client**
    - Solicitud de almacenamiento en cache (DEBUG)
    - Consulta de un valor en cache (DEBUG)
    - Operación solicitada, indicando:
        - Nombre del Archivo (DEBUG)
        - Tipo de operación (DEBUG)
        - En caso de lectura o escritura, el size y el offset (DEBUG)

- **Proceso Remote Cache**
    - Operación solicitada,indicando (DEBUG)
        - Key
        - Tipo de operación
    - Compactación para el esquema correspondiente (INFO)
    - En situaciones de reemplazo indicar algoritmo y key de la victima seleccionada (DEBUG)

Anexo B :: Protocolos de comunicación

Protocolo NIPC – (Network Inter-Process Comunication)

Este protocolo se utilizará para comunicar todos los procesos a través de paquetes. Un paquete de datos es una **unidad fundamental de transporte de información** en todas las redes de computadoras modernas. Los paquetes, al ser una unidad fundamental, no pueden ser divisibles. Por lo tanto, no se puede descomponer un paquete en varias partes y enviarlas por separado. Nuestro protocolo se maneja a nivel bytes, los cuales tienen el siguiente significado:



- ***Type:*** Identificador que determina el tipo de paquete que es. Esto hace referencia a la información que está contenida en el Payload.
- ***PayLoad Length:*** La longitud en bytes del campo siguiente, es decir del Payload.
- ***Payload:*** Este campo de bytes de longitud indefinida, contiene la información que se necesita enviar.

Extensión del Protocolo NIPC

El protocolo especificado anteriormente, especifica como deberá estar compuesto mínimamente cada mensaje, pero esto no indica que no puede ser ampliado. El grupo puede tomar decisiones sobre agregar nuevos campos al protocolo para facilitar su desarrollo. Esto cambios deben ser especificados en un documento y presentado al ayudante asignado durante el desarrollo y a los evaluadores cuando llegue el momento del coloquio.

Handshake

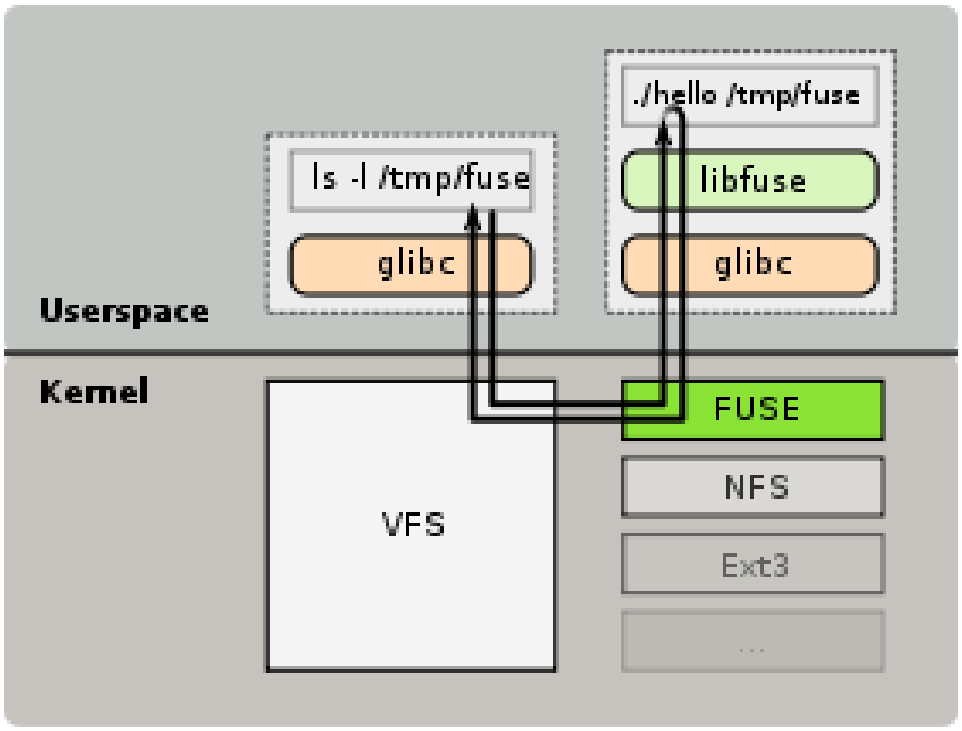
Handshaking ("apretón de manos") es un proceso de negociación que establece de forma dinámica los parámetros de un canal de comunicaciones establecido entre dos entidades antes de que comience la comunicación normal por el canal.

Para el trabajo práctico la funcionalidad del Handshake será la de negociar la comunicación entre dos procesos, haciendo que estos se identifiquen y se informen los parámetros iniciales.

Anexo C :: Introducción a FUSE

¿Qué es FUSE?

FUSE (Filesystem in Userspace) es un módulo de Kernel para sistemas operativos de tipo Unix, que permite crear sistemas de archivos sin necesidad de editar el código del Kernel. Esto se logra mediante la ejecución del código del sistema de archivos en el espacio de usuario, mientras que el módulo FUSE sólo proporciona un "puente" a la interfaz del núcleo real.



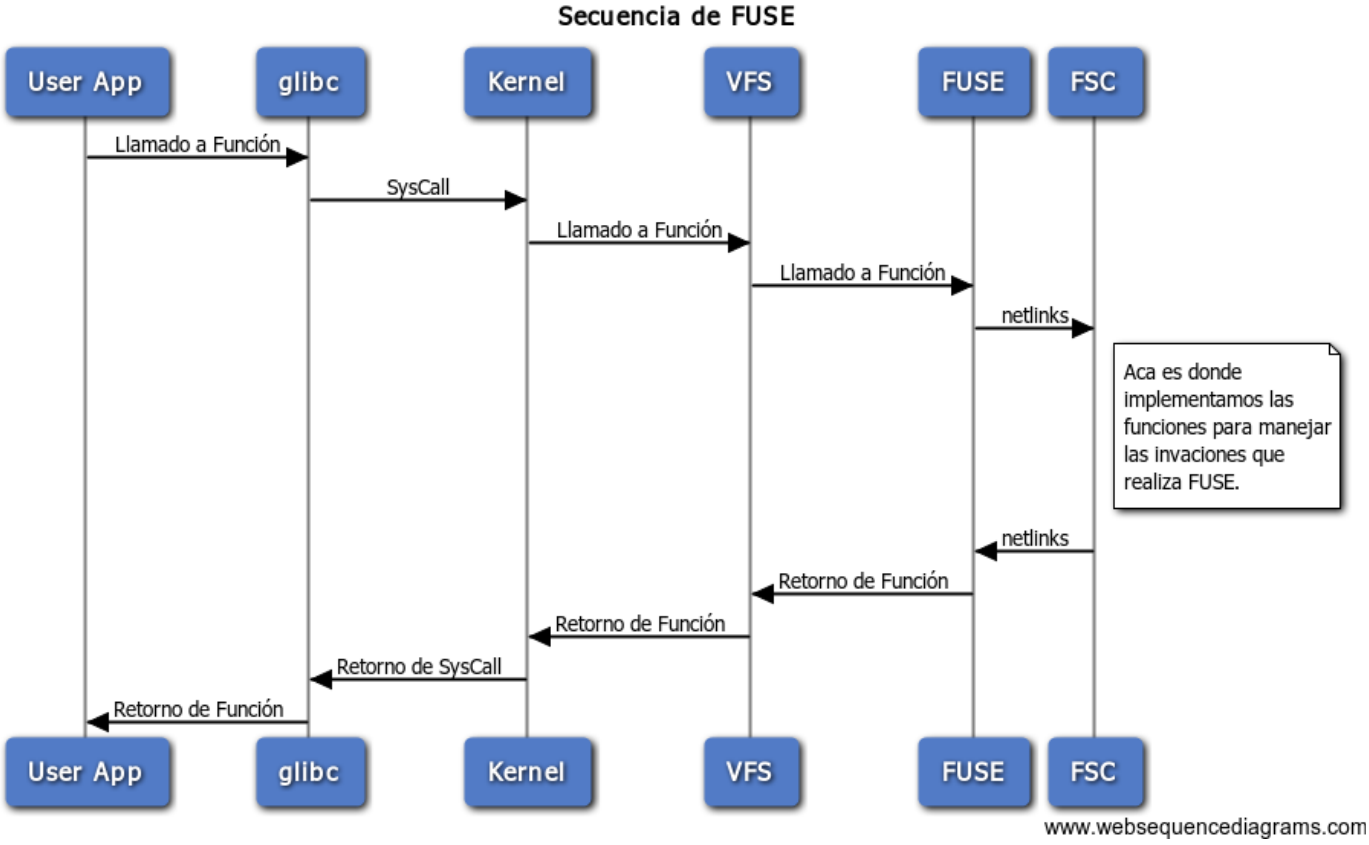
¿Como funciona?

Cuando desde el userspace se llama a una función de I/O, se termina invocando a la biblioteca del sistema la cual en algún momento termina generando una SysCall. Esta SysCall es capturada por el Kernel y delegada al VFS (Virtual FileSystem), la cual es una capa genérica a todas las posibles implementaciones de FileSystem que el Kernel poseea. Esta capa es la que se encarga de determinar, basándose en el path y los puntos de montaje, a cual implementación le corresponde resolver la operación de I/O (recordemos que dentro del FileSystem raíz podemos tener multiples FileSystem de diversos tipos montados). FUSE, al recibir la operación la delega al proceso que esté encargado de resolverla.

Internamente la biblioteca de FUSE trabaja con un mecanismo multi-thread, es decir, a cada operación se le asigna un nuevo thread de ejecución. Este mecanismo multi-thread puede ser deshabilitado pasándole como parámetro a FUSE ”-s”, opción que **simplifica la tarea de debuguear** (recordar que el TP será evaluado en un entorno multithread).

Otro de los mecanismos internos que posee FUSE es una cache interna propia. En la mayoria de las condiciones estas traen beneficios en cuanto a evitar accesos de I/O. Para el caso del TP **esta cache estará deshabilitada.**

Para anular la cache que posee la biblioteca de FUSE basta con pasar como parámetro la opción: ***-o direct_io***



Anexo D :: Introducción a Memcached

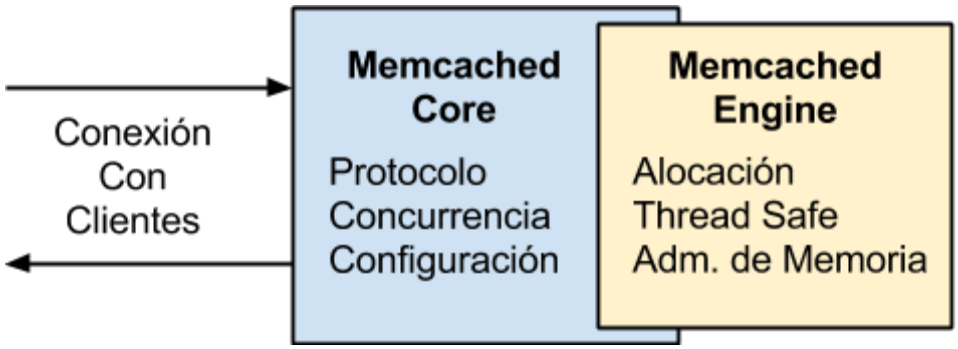
¿Qué es Memcached?

Memcached es un sistema de cacheo de objetos remoto de alta performance, libre y de código abierto.
Utiliza un formato del almacenamiento Key-Value, el cual es un esquema tipico de los sistemas NoSQL, al igual que como lo hace MongoDB, Berkeley DB, Redis o BigTable.

Arquitectura de Memcached

Memcached 1.6 posee una arquitectura mucho mas modular que sus versiones anteriores, y esto es porque el principal objetivo fue dividir a la aplicación en 2 partes:

- Core: Este es el núcleo de memcached, es el responsable de interpretar el protocolo, administrar los nuevos clientes y delegar las operaciones a los threads
- Engine: Esta parte es la que se encarga de administrar la información que se desea cachear en la memoria disponible. Esta consiste en una biblioteca compartida, la cual puede ser re-implementada para manejar el cacheo como más nos resulte conveniente.



Parámetros de Memcached

Los parámetros de inicialización más relevantes al trabajo son:

-E <path> Path a la shared library que funcionará como su engine.
-t <threads> Cantidad de threads a usar para atender los pedidos entrantes.
-p <port> Puerto por el que debe debe escuchar conexiones
-c <conn> Cantidad de conexiones que puede atender simultaneamente
-m <size> Tamaño de la cache en Mb.
-I <size> El tamaño de slab. Para el Esquema I, este valor se usará para indicar, en bytes,
el tamaño maximo de las particiones.
-n <size> El tamaño mínimo de chunk. Para el Esquema I, se usará para indicar el tamaño
minimo, en bytes, de las partición. Para el Esquema II indica el el tamaño

mínimo
: al que puede fragmentarse el Buddy System.

Anexo F :: Documentación

The Single UNIX® Specification, Issue 7 from The Open Group

http://pubs.opengroup.org/onlinepubs/9699919799/idx/headers.html

ext2

The Second Extended File system (ext2)
The Second Extended File System, Internal Layout. Dave Poirier.
The Linux Kernel, capítulo 9. David A. Rusling.
ext2 - OSDev Wiki

Hash

LibRHash

Memcached

Memcached - Intro
Memcached - Basic Protocol
Memcached - FAQ
How to writing your own storage engine for Memcached Part I
How to writing your own storage engine for Memcached Part II
How to writing your own storage engine for Memcached Part III
libMemcached

FUSE

FUSE Project

Se recomienda el siguiente material bibliográfico como soporte técnico:

- The C Programming Language de Kernighan & Ritchie
- Linux Programming Unleashed de Kurt Wall
- Linux System Programming: Talking Directly to the Kernel and C Library de Robert Love

Foro de Consulta:

Campus Virtual