Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería de Computadores

Trabajo Fin de Grado

Implementación de un sistema de ficheros

Autor: Mikel Cazorla Pérez

Tutor/es: Óscar López Gómez

2015-2016

Índice general

Índice de palabras

"Dear John:

I am a little troubled about the tea service in the electronic computer building. Apparently the members of your staff consume several times as much supplies as the same number of people do in Fuld Hall and they have been especially unfair in the matter of sugar.

Sugar is rationed and for a member of your staff to come up here as Thompson did and carry down a large quantity of sugar in excess of your rations is not cricket. I understand, furthermore, that the tea is served in several different places. We have never undertaken in the Institute to provide tea service in a large number of private offices and I should like to raise the question whether it would not be better for the computer people to come up to Fuld Hall at the end of the day at five o'clock in the afternoon and have their tea here under proper supervision. [...]"

Letter from the director to von Neumann (Institute for Advanced Study, Princeton).

"Querido John:

Estoy algo preocupado acerca del servicio de té en el edificio de electrónica de computadoras. Aparentemente los miembros de su personal consumen varias veces más suministros que el mismo número de personas en Fuld Hall y han sido especialmente injustos en el reparto de azúcar.

El azúcar es racionado y que un miembro de su personal venga aquí arriba como hizo Thompson y regrese transportando una gran cantidad de azúcar superior a vuestras raciones no es juego limpio. Entiendo, además, que el té es repartido en varios lugares. Nunca hemos emprendido en el Instituto el ofrecimiento de un servicio de té en un gran número de oficinas privadas y me gustaría plantear la cuestión de si no sería mejor para la gente de computadores ir a Fuld Hall al terminar su jornada a las cinco de la tarde y tomarse el té aquí bajo la adecuada supervisión. [...]"

Carta del director a von Neumann (Instituto de Estudios Avanzados, Princeton).

Abstract

This work consist of the development of a general purpose Plan 9™ class file system, able to run on unix-like operating systems[1], free and open.

In the field of operating systems, Plan 9 is highly remarkable, more than its predecessor (and when your predecessor is UNIX™, those are big words).

The resulting product pretends to be didactic and to serve as a research tool. Nevertheless, it is an operating system not very different to the UNIX one but simpler, designed to be extended on user space, and with distributed features.

In other words, anyone can consider using it. It is desirable that in addition to fullfilling its primary objetives will be useful for other developments and projects.

[1] Note the difference between *unix* and *UNIX*, where the first term includes those operating systems that resemble the original one, whose trademark to which refers the second initially belonged to Bell Labs. Overall, what is explained here about UNIX appliesto its clones and descendants, even when they were not fully compatible.

Resumen

Este trabajo consiste en el desarrollo de un sistema de archivos[2] de clase Plan 9™, de propósito general, capaz de funcionar sobre sistemas operativos unix[3], libre y abierto.

En el terreno de los sistemas de archivos, Plan 9 es un sistema altamente notable, más aún que su predecesor (y cuando tu predecesor es UNIX™, son palabras mayores).

El producto resultante pretende ser didáctico y servir como herramienta de investigación. No obstante, se trata de un sistema de archivos no muy distinto al de unix pero más simple, diseñado para extenderse en espacio de usuario, y de características distribuidas.

Dicho de otro modo, cualquiera puede considerar utilizarlo. Es deseable que, además de cumplir con sus objetivos primarios, sea útil para los de otros desarrollos y proyectos, dentro o fuera del entorno académico.

[2] En esta terminología, *sistema de ficheros* y *sistema de archivos* se utilizan indistintamente.

[3] Nótese la diferencia entre *unix* y *UNIX*, donde el primer término incluye a sistemas operativos que se parecen al original, cuya marca registrada a la que se refiere el segundo término pertenecía inicialmente a Bell Labs. En general, lo explicado aquí acerca de UNIX aplica a sus clones y descendientes, incluso cuando no sean del todo compatibles.

Capítulo 1

Introducción

UNIX™ es un claro ejemplo de como debe ser un sistema operativo. Durante medio siglo ha servido como material de estudio y ha sido clonado varias veces.

Para los que fermentan sistemas caseros es un modelo a seguir, también para empresas y universidades. Los sistemas de clase unix son apreciados en muchas áreas.

Lo que no es tan conocido es que, a mediados de los ochenta, la gente involucrada en su creación comenzó a desarrollar un nuevo sistema operativo de gran sencillez y carácter distribuido, que recibió el extravagante nombre de ***Plan 9™ from Bell Labs***[1].

UNIX, al fin y al cabo, surgió en el ocaso de la anterior generación, gobernada por enormes *máquinas de tiempo compartido*. En las que hasta cientos de usuarios trabajaban conectados mediante terminales, y cuyo precio estaba al alcance de empresas y universidades, pero no al de la gente corriente.

Desde entonces, una nueva generación había surgido impulsada por el **microprocesador** y las computadoras personales, y en una sola década había cambiado las reglas. UNIX se había adaptado para funcionar en estas máquinas, pero no se había diseñado para ellas.

Por otro lado, la evolución que tuvo desde finales de los sesenta lo había complicado. Siempre había sido identificado con el uso de programas pequeños y cohesivos, además de porque en él «todo *era* un fichero». Las nuevas características, como los gráficos o la red, se incorporaban al margen a su filosofía.

Esto puede parecer poco importante siempre y cuando funcionasen, pero tenía consecuencias prácticas, como la mala integración, una administración dificultosa, y la poca eficacia en el reparto de los recursos con respecto a los sistemas centralizados.

Plan 9 aprende de su predecesor, pero responde a estos problemas. Gracias a su actual licencia, además es **software libre**[1].

1.1 Motivaciones

En primer lugar, comprender la materia a nivel de diseño e implementación. En este caso, todo gira en torno a uno de los componentes principales de los sistemas operativos.

Plan 9 se caracteriza en esta área por unos principios de diseño ejemplares[2], aunque los alumnos no suelen llegar a oír jamás de el. El resultado permitirá iniciarse en su aprendizaje a estudiantes y entusiastas mientras lo utilizan cómodamente en sus distribuciones habituales.

También servirá como punto de partida para el desarrollo de un sistema operativo completamente funcional, y como campo de prueba para algunas de sus aplicaciones.

[1] Está disponible bajo licencia dual, una LPL y otra GPLv2.

[2] Es importante notar que *ser ejemplar* y *ser la única verdad* son cosas diferentes. El presente texto es posible en parte gracias a que se han estudiado y usado programas basados en filosofías contradictorias con intención de aprender. La experiencia ha sido muy productiva y agradable, por lo que se anima siempre a explorar otras perspectivas.

Capítulo 2

Marco teórico

Los sistemas operativos son anteriores a las computadoras personales, e incluso a los microprocesadores. Para alguien nacido a partir de los setenta, puede resultar difícil comprender en que consisten, incluso cuando todo el mundo habla de ellos constantemente.

Como indica Neal Stephenson en la introducción de su imprescindible ensayo «En el principio... fue la línea de comandos»[2], es llamativo como el individuo en la sociedad actual, cuando menos, ha adquirido unas turbias nociones acerca de los sistemas operativos, y con ellas ha forjado firmes opiniones.

Por ejemplo, circulan algunas sobre cuál es el mejor sistema operativo, o se escuchan otras acerca de la calidad de Windows™ 8. Sin embargo, cuando se trata de contestar a la pregunta *¿qué es un sistema operativo?*, las cosas ya no están tan claras.

Si se le pregunta eso mismo a diez personas, se obtienen diez respuestas diferentes. Cada respuesta es un intento de asignar una definición a la expresión *sistema operativo*, que desde hace mucho se ha convertido en un *término comodín*.

Pero en contra de lo que cabría esperar, tampoco hay unanimidad entre los «Morlocks»[1]. Esto supone un problema. Hacer un trabajo acerca de algo que ni siquiera se sabe lo que es no es una buena señal.

[1] Stephenson hace referencia a *La máquina del tiempo* de H. G. Wells mediante esta metáfora, en la que los *Morlocks* pertenecen a la clase social que comprende el funcionamiento de las cosas y es capaz de mover el mundo. En contraposición, los *Eloi* dependen de los *Morlocks* para sobrevivir.

Sería irresponsable seleccionar un libro de la materia y seguir sus definiciones sin advertir que —casi con seguridad— hay otras fuentes que lo contradicen a nivel terminológico. Esto no supone automáticamente que alguna este equivocada, aún cuando menos se están refiriendo a cosas distintas.

Para confundir más la cuestión, los estudiantes —nadie nace sabiendo— comienzan intentando encajar lo que van aprendiendo con esas «nociones imprecisas» que adquirieron siendo «Eloi», que en el mejor de los casos son superficiales, y a menudo, simplemente equivocadas.

A continuación se explica a que se refieren algunos términos *en los dominios de este trabajo*, y en concreto, que son los sistemas operativos y que tienen que ver con el sistema de archivos.

2.1 El sistema operativo

2.1.1 El monitor residente

Tradicionalmente, las máquinas no podían seguir las instrucciones de dos programas a la vez. De todos modos, de haber podido no habrían funcionado de la forma en la que se supone que debían hacerlo.

Se denomina **proceso** a la ejecución de las instrucciones de un programa para completar un objetivo. Que existan dos procesos al mismo tiempo (incluso si su programa es el mismo) implica compartir recursos, y para lograrlo, cada programa necesitaba ponerse de acuerdo con todos los demás. Esto estaba fuera de lo razonable.

En su lugar, los programas se ejecutaban en secuencia, uno tras otro. Y cada uno se leía desde un montón de tarjetas perforadas. Cuando un programa completaba su cometido, un operario debía recoger los resultados de la impresora y cambiar el montón por el de otro programa.

Pronto resultó evidente que casi todos compartían una parte del código, especialmente en lo que respecta a la entrada y salida. También, que organizar una cola de tareas es el tipo de cosa que a los ordenadores se les da bien.

En vez de hacer que todos los programas se pusieran de acuerdo con todos los demás, bastaba con que lo hiciesen con uno. Un programa especial que se cargaba antes que los demás. Varios montones de tarjetas se apilaban y, desde una computadora dedicada, eran previamente copiados a una cinta que era la que se utilizaba realmente como entrada.

Este **monitor residente** cargaba los datos y programas de la cinta, antes de ceder el control. Cuando un programa terminaba, cargaba al siguiente, hasta terminar. A menudo se aprovechaba para situar en él algunas rutinas de entrada y salida comunes, que los demás esperaban encontrar ahí, y que usaban a modo de biblioteca[2].

En síntesis, el monitor era un programa precargado que supervisaba la ejecución de cada **trabajo**. Por motivos obvios, su arquitectura solía adoptar un claro enfoque **monolítico**.

Por su manera de operar, la clase de máquina descrita se conoce como **sistema de procesamiento por lotes**, o **sistema *batch***[3]. De hecho, los resultados no se mostraban en una impresora, sino que se escribían como paso intermedio otra cinta, que otra máquina dedicada imprimía.

Es así como el computador que realizaba los cálculos no solo recibía los datos de entrada en una cinta, sino que devolvía otra como salida.

2.1.2 Sistemas multiprogramados

Distinguir entre *programa* y *proceso* no tenía mucho sentido en aquellos días, porque al fin y al cabo en el sistema *batch* solo había una configuración posible, y ambas cosas venían a ser parecidas.

Pero cada vez que un programa necesitaba realizar una operación de entrada y salida, debía esperar inactivo hasta obtener respuesta, así que se empezó a trabajar en una forma de aprovechar esas interrupciones.

[2]Esto puede resultar particulamente familiar a quienes han programado en ensamblador sobre MS-DOS™ o el BIOS, con respecto al vector de interrupciones.

[3]Nótese la ambigüedad en la palabra *sistema*, que en este caso se usa para hablar de la máquina completa como un conjunto de *software* y *hardware* (es decir, el computador), pero que también puede emplearse para referirse al *sistema operativo*. El sentido a lo largo de esta memoria se deducirá del contexto.

De pronto, los *residentes* comenzaron a repartir la memoria, y los programas se adaptaban a lo que se les asignaba en lugar de decidir ellos mismos donde situarse. De esta forma, se podía cargar más de uno. Cuando el monitor detectaba una espera de datos[4], este devolvía el control a otro trabajo.

También era importante llevar la cuenta del estado de la ejecución de cada programa. Dado que la «máquina desnuda» no incluía ningún mecanismo para seguir las instrucciones de dos programas, el monitor debía saber por donde continuar cuando interrumpía un trabajo y saltaba a otro.

2.1.3 El núcleo

De este primitivo mecanismo de **multiprogramación** al **sistema de tiempo compartido** solo hay un paso. Probablemente el que supuso un punto de inflexión entre el monitor residente y el **sistema operativo**.

Si se considera esta conexión evolutiva, el clásico programa monolítico conocido como **núcleo** es el sistema operativo. No obstante, lo que destaca no es su arquitectura, sino su rol en el computador.

Por un lado, es un **gestor** central que reparte los **recursos**. Los recursos pueden ser *hardware*, como la memoria (que se raciona en espacio) o el procesador (en tiempo), pero también pueden ser *software*, es decir, datos.

Por el otro, a menudo no los reparte de forma directa, sino que establece unas interfaces que los programas deben respetar. Más importante aún, esas interfaces describen un nivel superior, conceptos ficticios que ocultan a los auténticos recursos y sus particularidades, desacoplando la gestión de los mismos de los programas. Este tipo de artificios son llamados *abstracciones*.

Al trabajar como parte del computador, a ojos de los programas es parte del *hardware*. La máquina desnuda no es capaz de hacer muchas cosas por si misma, así que el sistema operativo es una capa que se interpone entre ambos extremos y añade las características necesarias. Cumple su función en forma de **máquina extendida**.

[4] Una manera hipotética de darse cuenta de cuando ocurrían tales esperas podría aprovecharse del intento de utilizar las rutinas de entrada y salida que incluía dentro.

Además, cuando el *hardware* que cuenta con los mecanismos de protección adecuados introduce el **modo usuario** y el **modo privilegiado** (o **modo núcleo**). En tales máquinas, el sistema operativo funciona en este último, teniendo acceso a ciertas características en exclusiva, facilitando su papel como supervisor y gestor de recursos. El resto de programas lo hace en el primero y por ello está restringido.

2.1.4 Dos núcleos y tres sistemas

Es por esto que en sistemas operativos de arquitectura **micronúcleo**, así como en los híbridos, se suele considerar que el núcleo es lo que queda en modo privilegiado, mientras que el resto de funciones pasa al modo usuario. Esto ha bifurcado la definición de núcleo: algunos llaman así al fragmento privilegiado, y otros al conjunto funcional. En el diseño monolítico, en cambio, no hay discusión porque ambos son la misma cosa.

Por otro lado, el sistema operativo se *distribuye* junto con algunos programas que se espera encontrar con el, en cualquier instalación. Por ejemplo, se espera que lo acompañe un **intérprete de órdenes** (o ***shell***[5]), un editor de textos o un programa de copia. A veces, cuando se habla del sistema operativo, se incluye este tipo de programas.

El término *UNIX* es empleado por la gente cercana al mismo de forma curiosa[3].

1. En sentido estricto, UNIX es el núcleo de un sistema operativo de tiempo compartido.
2. En un sentido algo más amplio, *también incluye* a algunos programas tales como editores, compiladores y otros que puedan considerarse necesarios.
3. En el más amplio de los sentidos, si alguien desarrolla un programa para ejecutarse en un sistema UNIX, este puede considerarse como parte de UNIX.

El problema es evidente al analizar el primer punto. A partir del mismo se puede interpretar que «en el más estricto de los sentidos, UNIX no es un sistema operativo».

[5]Los nombres *kernel* y *shell* forman una metáfora que hace referencia a las semillas.

Este es el caso si se considera que «el núcleo no es por sí mismo el sistema operativo», especialmente cuando UNIX es de arquitectura *monolítica*. De lo contrario, se producirían una serie de inconsistencias.

Como puede verse en algunas fuentes[4], también hay quien considera que en sentido estricto, el núcleo es el sistema operativo. La opción de este texto es tomar el camino de en medio, usando este último criterio para interpretar los tres puntos. Este no es probablemente a lo que se referían los creadores de UNIX exactamente, pero tampoco importa mucho, siempre y cuando que quede claro.

Además, supone que el sentido estricto no es el único correcto. El contexto determina a cual de los tres se hace referencia en cada caso.

2.2 El archivo

La palabra *archivo* se usa para designar un lugar donde se guarda información. En computadores, comenzó denotando a los soportes *hardware* que almacenaban dicha información. Por ejemplo, las unidades de disco en el IBM 350 eran llamadas *archivos de disco*. En este caso puede entenderse literalmente.

Pero actualmente no es el significado más frecuente. Los sistemas operativos deben conseguir que los programas obtengan y compartan almacenamiento. Pero no ceden espacio de disco como tal, sino que imponen una de sus abstracciones.

2.2.1 Concepto abstracto

Un archivo puede describirse como una cinta imaginaria, con la que se interactúa mediante **tres operaciones** básicas. Se puede **leer** o **escribir** desde una posición de la misma. Estas operaciones avanzan la posición sobre la cinta automáticamente, aunque existe una tercera para (re)**ubicarse** en otra posición antes de proseguir con las otras dos.

En cierto tipo de archivos, esta última operación no esta disponible y no se puede gobernar la posición en la cinta. Se suele hablar de archivos en **modo carácter**, para diferenciarlos de los de **modo bloque**.

2.2.2 Acceso secuencial

A menudo se dice que los de modo bloque se diferencian de los otros por ser de acceso aleatorio, aunque esto no es correcto. Las cintas se consideran el ejemplo por excelencia de acceso secuencial. La tercera operación se ocupa de controlar un cursor imaginario, impropio en un dispositivo de acceso aleatorio.

Un buen ejemplo son los discos duros de estado sólido. En los clásicos discos mecánicos, debido al uso continuado, los fragmentos de las estructuras de datos se dispersaban y desordenaban en el interior del disco.

Todo estaba ahí, pero debido al acceso secuencial, las lecturas y escrituras no incidían en *sectores* contiguos, lo que obligaba al cabezal a realizar más movimientos. Esto vuelve la interacción mucho más lenta, porque para saltar a las distintas posiciones, siempre hay un trayecto intermedio a recorrer.

La infame *fragmentación* del disco se pude contrarrestar recolocando los *bloques* para aprovechar mejor las pasadas mediante un acceso contiguo. La desfragmentación no debe añadir ni borrar los datos, así que es un proceso singular, ya que realiza lecturas y escrituras, pero ningún trabajo. Todo para mejorar el degradado tiempo de respuesta.

Sin embargo, los discos de estado sólido proporcionan acceso directo, ya que no tienen partes mecánicas móviles. O lo que es lo mismo, acceder a un bloque no requiere atravesar una serie de puntos intermedios *desde la última posición*. Aunque hay otra clase de fragmentación de la que preocuparse, esta en concreto no afecta al rendimiento e intentar combatirla es inútil[6], e incluso prevenirla puede resultar contraproducente[7].

2.2.3 Toda abstracción es sintética

Así, cuando un proceso interactúa con un archivo, puede hacerlo como si leyese o escribiese sobre su propio rollo de cinta dedicado[8]. Pero esto solo es una abstracción.

[6] Sumado a la escasa vida útil de los primeros discos de esta clase, es una buena manera de echarlos a perder.

[7] Algunos **formatos del sistema de archivos** (explicados en la sección correspondiente) tienen en cuenta la fragmentación en su diseño.

[8] Esto es una primera aproximación. Obviamente, también pueden dos procesos intentar operar sobre el mismo archivo concurrentemente, lo que conlleva su propia problemática.

La RAM pierde los datos tras el apagado, así que se requiere un mecanismo que permita preservarlos. Esto se denomina **almacenamiento persistente**, y para proporcionarlo se utilizan diversos dispositivos, p. ej., disquetes, discos duros, CDs, pendrives, etcétera.

Los datos generados por los procesos pueden consignarse en uno de estos dispositivos de almacenamiento multiplexado por el sistema operativo, cuando estos creen disponer de una o más cintas dedicadas.

Otra ventaja es que se puede pasar el resultado de un programa a otro directamente, como en una cadena de montaje. Para ello se usa una **tubería** para conectar dos programas, que por lo que a los programas respecta es un archivo, pero el contenido no se almacena persistentemente, sino que le es hecho llegar al lector con ayuda de un *buffer*.

En definitiva, el que proporciona esta abstracción es el sistema operativo, y por eso puede decidir si guardar unos *bytes* en un disquete, o hacer cualquier otra cosa. También puede afirmar la existencia de un archivo que no existe, y lo contrario. Es más, un archivo existe porque así lo dice el sistema operativo.

Cuando un archivo no se almacena de la forma en que debe en el dispositivo de almacenamiento, se le suele denominar **archivo sintético**, haciendo referencia a que no es un archivo «de verdad». Por ejemplo, en los sistemas de clase unix, hay un archivo que representa al disco duro, y este se puede leer (y escribir) directamente.

Sin embargo, no se puede decir que el archivo que representa al disco duro no haga lo que se supone que debe hacer. Es más, los archivos «reales» no tienen nada de real. Una forma de verlo es que todos deben ser sintetizados por el sistema operativo. Al fin y al cabo, la expresión *archivo sintético* es solo una manera de hablar.

2.2.4 El formato del sistema de archivos

Se conoce como **formato del sistema de archivos** a las estructuras de datos empleadas para almacenar persistentemente el contenido de los archivos, junto con los metadatos necesarios, lo que permite su posterior recuperación. Algunos ejemplos de estos formatos son EXT2, EXT4, kfs, Fossil, FAT32, NTFS, HFS+, BeFS, ZFS, etcétera.

Incluso leer unas estructuras EXT o FAT requiere cierto grado de interpretación por parte del sistema operativo. Esta matización era irrelevante en aquellos sistemas que solo utilizan el concepto de archivo de manera «no sintética». A estos formatos también se les llama, simplemente, *sistemas de archivos*. Pero esta definición no es el centro de este trabajo.

2.2.5 El sistema de archivos

Comprender en que se diferencian los archivos «sintéticos» (o mejor, por qué no se diferencian) es necesario para entender la frase «en UNIX, todo es un archivo». Eso significa que alguien procedente de otra clase de sistema tendría otro concepto de *archivo*, y se preguntaría el motivo por el que nadie querría eso.

El problema que afrontaba el **almacenamiento de datos compartido** se solucionó cuestionándose **como los programas se comunican**, y ofreciendo una implementación inspirada en el mecanismo de cinta. Este enfoque se puede utilizar para afrontar otros problemas, así que todo se reduce a si el sistema de archivos se define en base al problema original, o en los principios con los que este se aborda.

Por eso existen dos definiciones para *sistema de archivos*. La segunda denota todo lo que está en manos del sistema operativo para proporcionar esa abstracción y dar servicio. Se podría argumentar que la primera se engloba en esta, aunque teniendo en cuenta que su presencia no es esencial: los archivos en sistemas unix se usan para una gran cantidad de cosas, y seguir siendo útiles haya o no persistencia.

Plan 9, además de partir de los mismos fundamentos es un sistema distribuido. Tómese como ejemplo. Una máquina sin disco duro puede emplearse como terminal, y a pesar de ello los programas, los procesos, la memoria, las máquinas en red, y prácticamente cualquier cosa imaginable es un archivo.

Es difícil imaginar que la más simple de las sesiones no pase por el uso de un buen puñado de ellos, incluso si esta no implica el uso de un servicio de almacenamiento, local o remoto.

2.2.6 Propósito del «todo es un archivo».

Aceptar este hecho y comprenderlo son dos cosas distintas. Primero, hay una gran variedad de recursos que se manejan bastante bien mediante una fachada con forma de archivo. La evidente es compartida con los otros sistemas: permitir almacenar en un disco duro un puñado de datos, compartiendo los mismos y el espacio con otros procesos.

Pero *leer* y *escribir* son dos operaciones muy comunes. Con ellas puede cambiarse la configuración de los programas y el sistema, dirigir un componente *hardware*, o comunicarse a través de una red. Incluso los sistemas unix incorporan en su API porciones específicas para realizar muchas de estas tareas.

Normalmente estos fragmentos de API incorporan, al menos, esas dos operaciones en alguna variedad (a menudo acompañadas de un *espacio de nombres* y un esquema de seguridad). Hasta los dispositivos de acceso aleatorio pueden manipularse bajo una interfaz secuencial, así que uno de los problemas con este tipo de APIs es que suelen ser redundantes.

Lo peor es que han sido pensadas para no ser tan generales, y eso tiene consecuencias. Puede imaginarse que el disco duro no sea un archivo, y que manipularlo implique utilizar otra API diferente (por ejemplo, la del sistema de entrada y salida). Se puede escribir un programa de copia archivo-archivo, y otro disco-disco. Pero si se quiere copiar a un archivo el contenido de un disco y viceversa, se necesitan otros dos programas, haciendo un total de cuatro.

Si se añade al cálculo otra abstracción con su interfaz correspondiente, se asciende a nueve programas, en una progresión geométrica. Básicamente, estos realizan operaciones de lectura que vuelcan mediante otras de escritura, pero la función de programa correspondiente a cada interfaz es distinta. Salvo eso, el código de los nueve programas es idéntico: un «copiar y pegar» (de código, en este caso).

Si cada abstracción compartiese API, el problema se podría solucionar con un solo programa. Otro beneficio es que una nueva abstracción no implica reconstruirlo ni alterar su código. Esto permite usar ese programa para cosas y de maneras en las que no se pensaba al diseñarlo, pero para las que resulta muy conveniente. Por último, para el programador es mucho más fácil usar una API con una sola variante de las operaciones básicas, que recordar una miríada de versiones específicas.

En un sistema operativo suele haber algo más que un programa de copia, así que esto parece importante.

2.2.7 El espacio de nombres

Hay otras dos operaciones importantes a la hora de trabajar con archivos. Queda claro que los programas pueden trabajar con ellos, pero en un sistema lo normal es que existan más de uno y de dos, a diferencia de los sistemas *batch*. A veces incluso deben compartir un solo archivo. Se requiere otro mecanismo para «seleccionar la siguiente cinta».

Para distinguir un archivo de otro, estos se asocian con un identificador único[9], en este caso un **nombre** en el sistema. Para comenzar a utilizar un archivo, es posible **abrirlo** a través de su identificador, lo que equivale a colocar la cinta. Para finalizar su uso se emplea su contraparte: **cerrar**.

A diferencia del mecanismo de una única cinta, donde no había que distinguir cuál se deseaba utilizar, esta abstracción se basa en **asignar** nombres a cada archivo. Esto significa que estos son accesibles en cuanto están asignados a un nombre único, y a través del mismo.

El conjunto de nombres asignados a archivos, por tanto, es conocido como espacio de nombres.

2.2.8 El árbol de directorios

Como los programas pueden emplear más de un archivo, también pueden mantener diferentes grupos de datos separados y clasificados en el espacio de nombres. En un espacio de un solo nivel esto puede ser complicado, porque no puede haber dos archivos con el mismo nombre. Además, con el paso del tiempo cada vez hay más... contribuyendo al desorden y extravío.

Una estrategia común es permitir la creación de *subespacios de nombres*, en una jerarquía donde mantener semánticamente[10] agrupados a aquellos archivos que estén relacionados.

[9] Debe tenerse en cuenta que hay diferencia entre *identificador único* y *único identificador*.

[10] Aunque no en el sentido de los *sistemas de ficheros semánticos*, que (actualmente) no gozan de popularidad en el ámbito de los de propósito general, y no se analizan aquí.

Así, un espacio de nombres es un **directorio**[11]. No todos los sistemas abordan una solución de esta clase de la misma manera ni usan la misma terminología, algunos incluso emplean un número de niveles fijo junto o una sintaxis específica. No obstante, se pone como ejemplo el modelo unix.

El mecanismo empleado para manejar los espacios de nombres tiene un parecido (remoto) al descrito en MS-DOS 1.x. Mediante un carácter separador (en unix se utiliza la barra oblicua sin invertir, «/»), se concatena el nombre de cada subespacio. Eso hace que el nombre «efectivo» de un archivo sea diferente al que se le asignó directamente bajo el **directorio padre**. Este nuevo nombre se llama **ruta**.

vacaciones/2016/07/09-34.jpg

Que, obviamente, es la ruta a una de las fotografías de las vacaciones. Si los aficionados a la fotografía tuviesen que usar un espacio de un nivel y tuviesen solo ocho caracteres más extensión, los nombres razonables empezarían a escasear con alarmante rapidez. Por ejemplo.

16070934.jpg

El nombre sigue representando la fecha, aunque con cierta dificultad. El problema mayor es que ahora no solo compite con otras fotos de las vacaciones, sino con cualquier otro posible archivo que desee usar la fecha como nombre.

Por ejemplo, una captura de pantalla de un fallo ocurrido durante esa hora. En el caso de los directorios, esto no es problema porque se hospedaría en un subespacio separado, con un nombre apropiado.

bitácora/2016/07/09-34.jpg

Pero en lugar de acceder desde el espacio de nombres «primario», se puede seleccionar un subespacio como directorio activo[12], de la misma manera en que se puede cambiar la unidad activa en MS-DOS. Siguiendo con el ejemplo, si tal directorio fuese «vacaciones/2016», la ruta del primer ejemplo sería distinta:

07/09-34.jpg

[11] En MS-DOS se usa este término, probablemente por herencia de CP/M, del que aparentemente también toma la orden «DIR». A diferencia de estos, en Windows también se usa «carpetas» para referirse a un concepto más general, que incluye al equivalente del primero de contenido sintético. En sistemas unix se habla simplemente de directorios.

[12] También llamado **directorio de trabajo**.

Esto es, las nombres visibles son los que pertenecen al espacio activo, condicionando a su vez a las rutas disponibles. También se puede cambiar al del mes correspondiente:

09-34.jpg

En el intérprete se usa la orden «cd» (o «change directory»), que invoca a una *llamada al sistema*: «chdir».

cd vacaciones/2016/07
Ahora el directorio actual es vacaciones/2016/07.

Como se ha explicado anteriormente, el carácter separador está reservado y no puede utilizarse en los nombres. De lo contrario, el sistema se «sentiría confuso» por ambigüedad («¿eso es una carpeta dentro de otra, o el archivo se llama así?»).

En ocasiones es necesario hacer referencia al propio directorio, o ascender al directorio padre. Para ello, todos contienen dos nombres reservados: «.» y «..». De esto se deduce que los archivos tienen **varios** identificadores únicos. Para acceder a la misma fotografía, las siguientes rutas son equivalentes:

vacaciones/2016/07/09-34.jpg
vacaciones/2016/../2016/07/09-34.jpg
vacaciones/2016/07/./09-34.jpg

Aunque parece que estas rutas no tienen utilidad, un programa cuyo directorio de trabajo sea «vacaciones» puede acceder a la imagen de la bitácora antes mencionada:

../bitácora/2016/07/09-34.jpg

Una última consideración, es que esta estructura tiene un nodo «primario», sin padre[13]. Para acceder directamente al espacio de nombres primario, hay otro nombre siempre disponible, que también es especial ya que ningún otro archivo puede llamarse así.

/

Además de cambiar rápidamente a este directorio, se puede acceder (sin importar el directorio activo, ya que no hay ambigüedad) a cualquier archivo mediante su **ruta absoluta**[14]:

[13] En realidad, el directorio «..» en el nodo primario es el mismo, así que es su propio padre. Este matiz no cambia el razonamiento descrito a continuación.

[14] Las de la otra variedad se llaman **rutas relativas**.

/vacaciones/2016/07/09-34.jpg
/bitácora/2016/07/09-34.jpg

La estructura resultante es un **árbol jerárquico**.

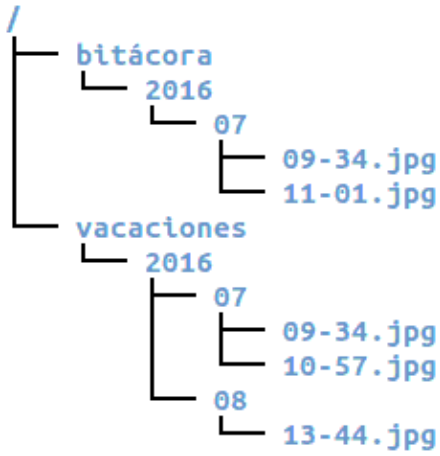


Figura 2.1: Ejemplo de árbol de directorios, de hasta cuatro niveles.

Al directorio identificado como «/» se le suele llamar **raíz**.

2.2.9 Permisos[15]

En sistemas unix, el concepto es realmente sencillo y eficaz. Cada usuario está identificado en el sistema, y tiene que autenticarse para iniciar una sesión en él.

Desde ese mismo momento, los procesos que va creando están asociados a su usuario, así que cuando uno de ellos intenta interactuar con un archivo tan solo hay que comprobar si este tiene permiso para realizar esa acción.

Cada archivo tiene un dueño y una secuencia de nueve bits (normalmente representados en octal) divididos en tres grupos: los permisos del poseedor, de su grupo y de los demás. Los bits de cada grupo representan el permiso de lectura, escritura y ejecución respectivamente («rwx»).

[15]La seguridad es muy importante, sin embargo este apartado no puede abordarse por completo por una limitación en el sistema de autenticación. La solución requiere otro módulo del sistema operativo, algo que queda fuera del ámbito de este trabajo. Tal vez en el futuro...

De esta manera, un archivo que solo permite la escritura a *su* usuario, pero que admita la lectura y ejecución a todo el mundo, tendrá una combinación de permisos como esta: «rwxr-xr-x». O lo que es lo mismo, «0755»[16].

Un usuario que desconfía de si mismo podría evitar modificar un archivo propio accidentalmente mediante una combinación de permisos «r-xrwxrwx» (0577). O, por ejemplo, se puede permitir en exclusiva la lectura del archivo mediante «r–r–r–» (0444).

Además, los permisos funcionan diferente con los directorios: el permiso de ejecución se convierte en el de búsqueda, y permite «caminar» a través de él. La lectura permite ver su contenido. Jugar con esto puede dar lugar a situaciones interesantes (¿y si se tiene el permiso de búsqueda, pero no el de lectura?).

```
/inside % chmod -r test
/inside % ls test
ls: no se puede abrir el directorio test: Permiso denegado
/inside % cd test && echo Success
Success
/inside/test % ls
ls: no se puede abrir el directorio .: Permiso denegado
/inside/test %
```

Por supuesto, el permiso de escritura afecta a la modificación de que archivos se encuentran dentro. Ha de tenerse cuidado, sin embargo, porque este no se propaga a los subdirectorios que pueda contener («apuntar»), y es posible que aunque el padre tenga ese permiso desactivado, dentro de estos todavía sea posible crear, modificar y borrar archivos.

[16] Cuando un número en lenguaje C empieza por cero, significa que está representado en octal.

Capítulo 3

Estado del arte

3.1 MS-DOS

MS-DOS™[1] es un sistema operativo desarrollado por Microsoft durante unas dos décadas, desde principios de los ochenta.

Las primeras versiones de Windows eran una extensión sobre el MS-DOS clásico, que luego fue siendo alterado. Esta rama fue finalmente desplazada en favor de NT, si bien ambos siguen conectados en muchos sentidos.

3.1.1 MS-DOS 0.x, ejemplo práctico

Aunque tal vez hubo una versión así numerada (o al menos, del sistema que sirvió de prototipo), en este texto se usará para referirse a una *hipotética* (e improbable) versión primigenia, predecesora de las que salieron al mercado bajo la numeración 1.x. Un inicio de sesión podría ser así[2]:

[1] También conocido como PC-DOS en los IBM PC genuínos, hasta la bifurcación de la versión 6.1, en 1993.

[2] Las siguientes imágenes han sido ligeramente modificadas por legibilidad.

```
MS-DOS Command release 0.00
Current date is Tue  1-01-1980

:DIR
COMMAND  COM     4986   1-18-84   2:01p
CHKDSK   COM     1754   8-16-83  12:56p
CONFIGUR COM    19724   5-03-84  10:33a
DEBUG    COM     6003   8-16-83   1:02p
DISKCOMP COM     5344  11-11-83   1:37p
DISKCOPY COM     5728  12-13-83   1:37p
EDLIN    COM     2313   8-16-83  12:56p
EXE2BIN  EXE     1280   8-16-83   1:01p
FILCOM   COM     8320   8-16-83  12:56p
FORMAT   COM     3856   4-24-84   3:50p
LIB      EXE    32128   8-16-83   1:00p
LINK     EXE    41856   8-16-83   1:00p
PRINT    COM     1740   8-16-83   1:43p
RDCPM    COM     3548  12-13-83   1:28p
SYS      COM      914   1-18-84   2:50p
TEST     BAK       29   1-01-80   2:19p
TEST     TXT       57   1-01-80   2:21p
       17 File(s)

:_
```

En azul se muestra lo que ha escrito el usuario. «DIR» es una orden de MS-DOS que muestra los archivos disponibles en el espacio de nombres. Hoy día puede ser extraño, pero este espacio de nombres solo tiene un nivel, así que estos son todos los archivos, y no hay forma de agruparlos para clasificarlos.

Parece sorprendente que se pueda operar así, dado que no se podían crear dos archivos con el mismo nombre (de ocho caracteres máximo) en todo el sistema, y en especial con las necesidades actuales. Pero a principios de los ochenta, la industria no estaba tan desarrollada en el mercado doméstico, y MS-DOS evolucionó con rapidez.

En rojo se muestra un cursor parpadeante. Tanto a la izquierda de la orden como a la del cursor aparecen dos puntos «**:**», en el borde de la pantalla. Eso es el **indicador**, o ***prompt***, que sirve para que el usuario sepa que el intérprete está a la espera de que el usuario introduzca una nueva orden.

3.1.2 MS-DOS 1.x y los múltiples espacios de nombres

El auténtico MS-DOS no es muy diferente. Una sesión real puede ser así:

```
MS-DOS Command release 1.00, version 1.19
Current date is Tue 08-31-1981


A:DIR
COMMAND  COM     4986   1-18-84   2:01p
CHKDSK   COM     1754   8-16-83  12:56p
CONFIGUR COM    19724   5-03-84  10:33a
DEBUG    COM     6003   8-16-83   1:02p
DISKCOMP COM     5344  11-11-83   1:37p
DISKCOPY COM     5728  12-13-83   1:37p
EDLIN    COM     2313   8-16-83  12:56p
EXE2BIN  EXE     1280   8-16-83   1:01p
FILCOM   COM     8320   8-16-83  12:56p
FORMAT   COM     3856   4-24-84   3:50p
LIB      EXE    32128   8-16-83   1:00p
LINK     EXE    41856   8-16-83   1:00p
PRINT    COM     1740   8-16-83   1:43p
RDCPM    COM     3548  12-13-83   1:28p
SYS      COM      914   1-18-84   2:50p
TEST     BAK       29   1-01-80   2:19p
TEST     TXT       57   1-01-80   2:21p
\\\                 1   8-31-81   6:08p
       18 File(s)

A:_
```

La única diferencia es que el *prompt* ahora se compone también de una letra. El motivo es que si bien los archivos del espacio de nombres se pueden preservar en un solo *disquete*, resulta inconveniente no poder transferirlos a otro secundario.

Al principio, los IBM PC no tenían disco duro, así que trabajaban con disqueteras (*floppy drives*[3]) y disquetes (*floppy disks*), y que era para lo que estaba pensado MS-DOS inicialmente. Había modelos con dos disqueteras, por lo que siempre existía la posibilidad de que dos archivos tuviesen el mismo nombre. Por tanto, no podían existir en el mismo espacio.

[3]Con respecto a la nomenclatura en inglés, el término *unidad de disquete* es más parecido, y en este caso, conveniente.

Otro problema relacionado era que si se deseaba crear un archivo nuevo (o copiar uno existente) había que indicar de alguna manera sobre que disquete debían actuar las operaciones.

La solución adoptada[4] fue que en el arranque se asignase una letra a cada «unidad de disquete»[5]. De esta manera, cada letra identificaba (durante el funcionamiento) un espacio de nombres diferente.

En circunstancias normales, no se debía hacer nada en especial, así que el sistema mantenía la retrocompatibilidad con los programas de la versión anteriormente imaginada. El espacio de nombres asociado a la letra del *prompt* era el que estaba activo, así que no había que preocuparse por colisiones de nombres.

De pronto, el usuario decide editar un archivo llamado «README» en el espacio de nombres de otro disquete. Este proporciona una pista mediante un prefijo al escribir el nombre del archivo:

```
A:EDLIN B:README_
```

EDLIN es el editor de textos de MS-DOS. Si el usuario escribiese «EDLIN README», el archivo a editar sería buscado en el espacio de nombres actual, asignado a la letra «A».

Antes de usar un programa, es posible cambiar a otro espacio de nombres:

```
A:B:

B:_
```

Lo mejor de este mecanismo es que los programas no tienen que hacer nada que no hiciesen en la versión 0.x. Ni siquiera se molestan en comprobar el nombre del archivo[6], que reenvían del modo habitual al núcleo de MS-DOS. Este es el que advierte la presencia del prefijo y actúa en consecuencia.

Por desgracia, también tiene una serie de inconvenientes como se explica más adelante, y por los que se aconseja evitar este enfoque.

[4] En realidad, adquirida de CP/M™.

[5] En Windows no solo se pueden asociar con unidades (también sirve con particiones, e incluso directorios), pero el término se sigue usando por herencia de MS-DOS, lo cual es motivo de confusión.

[6] Aún así puede ser necesario asegurarse de que no son «opciones» (también conocidas como *flags* o *switchs*).

Como puede comprobarse, hay un archivo llamado «\\\». El carácter de la barra oblicua invertida suele estar reservado y no puede utilizarse dentro del nombre de un archivo, pero aún en estas versiones de MS-DOS estaba disponible.

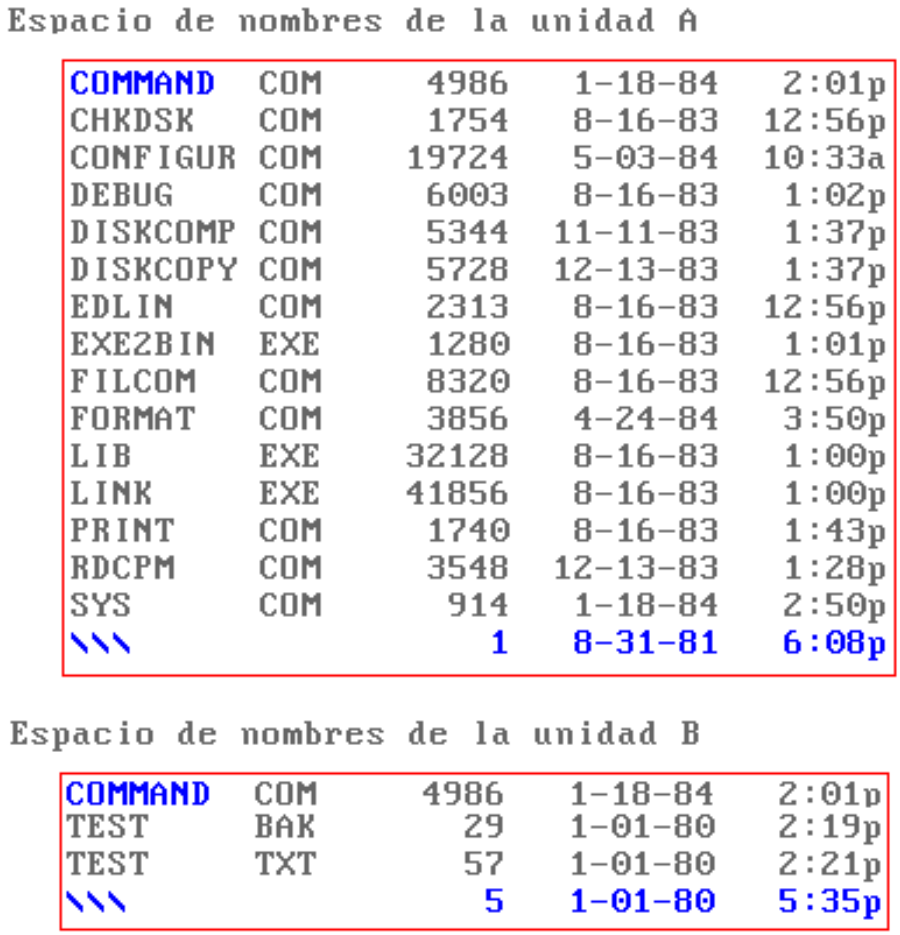


Figura 3.1: Este ejemplo muestra que nombres hay activos. Cada espacio de nombres (unidad) representa un disquete. El contenido no puede estar en un mismo espacio porque no se podrían distinguir archivos diferentes con el mismo nombre.

3.1.3 Desde MS-DOS 2.x en adelante

Las versiones 1.x, tal como se han descrito en la sección anterior, no saben lo que es un «directorio», ni siquiera uno raíz. Aún así utiliza el término para referirse a una de las estructuras de datos en las que se dividía el «espacio de disco»[7].

Esta limitación ya no estaba presente en la 2.x, así a partir de esta las sesiones resultan más familiares[8].

```
MS-DOS(R) Version 4.01

C:\DOS>mkdir HOLA

C:\DOS>cd HOLA

C:\DOS\HOLA>DIR

 Volume in drive C is DOS400
 Volume Serial Number is 1231-1508
 Directory of  C:\DOS\HOLA

.            <DIR>      08-27-16   2:05p
..           <DIR>      08-27-16   2:05p
        2 File(s)  521306112 bytes free

C:\DOS\HOLA>cd ..

C:\DOS>DIR | MORE

   (Parte de la salida omitida por conveniencia)
DOSUTIL  MEU       6660 04-10-89   9:00a
SHELL    CLR       4406 04-10-89   9:00a
SHELL    HLP      66527 04-10-89   9:00a
SHELL    MEU       4588 04-10-89   9:00a
SHELLB   COM       3894 04-10-89   9:00a
SHELLC   EXE     153855 04-10-89   9:00a
HOLA          <DIR>      08-27-16   2:05p
       66 File(s)  521306112 bytes free

C:\DOS>_
```

[7] De esa manera se expresaba el manual, que aunque no especifica, seguramente se refiere al espacio del *disquete*.

[8] Se ha usado en las imágenes la version 4.x en lugar de una 2.x o 3.x por la claridad del intérprete, a pesar de que estas ya incorporan el árbol de directorios.

Como puede apreciarse, el *prompt* es muy diferente. Hasta ahora, se mostraba la letra de la unidad cuyo espacio de nombres estaba activo (recuérdese el uso del carácter «:» como separador). Esta vez, también muestra la ruta absoluta del directorio de trabajo. Si en cualquiera de los dos casos se omitiesen esas indicaciones, el usuario podría olvidarse y, accidentalmente, interactuar con los archivos equivocados.

Anteriormente, el carácter «:» era el símbolo que se empleaba para marcar donde termina el *prompt*[9], y no era necesario usarlo como separador (puesto que excepcionalmente no actuaba como prefijo). Ahora que eso ha cambiado, y ese carácter separa la letra del resto de la ruta, se utiliza en su lugar «>» como finalizador del *prompt*.

```
A:

C:\DOS\HOLA>
```

Figura 3.2: El símbolo «:» ya no se utiliza de la misma manera, así que «>» ocupa esa función.

El carácter «\» es el otro separador, y a la vez se usa como el nombre del directorio raíz. Por este motivo, ya no puede usarse para crear nombres de archivo.

```
A:

C:\DOS\HOLA>
```

Figura 3.3: Hay dos separadores (en azul), cada uno con una misión. El símbolo «/» cobra otro sentido si usa como identificador del directorio raíz (en rojo). Antes no se usaban separadores en el prompt.

Además, este sistema en concreto está instalado en un disco duro. Como puede comprobarse, MS-DOS prefiere empezar a asignar letras a estos dispositivos a partir de la «C», en referencia al uso en sistemas de dos disqueteras.

De todo esto se pueden extraer algunas conclusiones. En primer lugar, el árbol de directorios es extremadamente útil, y sin embargo, no imprescindible en un sistema de ficheros. Durante mucho después del lanzamiento de UNIX, si se tenía la suerte de disponer de uno, a veces se implementaba de maneras muy pobres, con limitaciones que hasta el usuario más dispuesto no pasaría por alto hoy día.

Por otro lado, se da la paradoja de que el árbol de directorios de MS-DOS (y por tanto, de Windows) está inspirado en el de unix, aunque no contaba con el en las primeras versiones. El sistema de ficheros básico y los primeros programas provienen de los de CP/M. Probablemente debido a que en este los *switchs* utilizan el símbolo «/» en lugar de «-», se optó por la barra invertida para sustituir al separador.

[9] **Solo** en el *prompt*. Al utilizarse como prefijo en órdenes y programas hace de separador con normalidad.

Las desventajas del uso de múltiples árboles de directorios están directamente relacionadas con las de las letras de unidad en MS-DOS. Mientras que se suele alabar el uso de un único árbol de directorios de unix frente al «enfoque MS-DOS», en realidad este último es un efecto secundario de combinar dos estrategias de gestión de espacios de nombres, siendo una de ellas, en muchos sentidos, parecida a la de unix. Se sugiere como ejercicio introducir rutas absolutas sin prefijo en un sistema MS-DOS/Windows.

No han sido pensadas para trabajar simultáneamente (se mantiene la primera por retrocompatibilidad), y son en cierta medida redundantes: una unidad, partición o fichero que contenga un formato del sistema de archivos requiere conectarse a un espacio de nombres para poder utilizarse, y tanto las letras como los directorios pueden cumplir con ese cometido.

El sistema funciona bien bajo uso adaptativo en tiempo de ejecución. Mientras los usuarios puedan «sugerir» donde se encuentra un archivo, y los programas no hagan uso de ese poder, no hay problema. Pero si los *scripts* y programas incluyen el prefijo en su interior, puede ocurrir:

1. Que durante el siguiente arranque las letras se asignen de otra manera. Pueden desde aparecer dificultades menores, a que los programas dejen de funcionar.
2. Que los programas exijan que nunca se cambie el número de unidades ni el rol de cada una.
3. Que los programas exijan un número de unidades y sus roles predeterminados por el fabricante del mismo, *negándose* a funcionar en cualquier otra máquina.

3.2 Unix

Incluso a pesar de no llevar a sus espaldas la carga de un sistema de espacios de nombres obsoleto y redundante, unix afronta problemas similares.

En primer lugar, los *montajes de unión*[10] mantienen al *hardware* acoplado a la estructura de directorios (basada en identificadores semánticos), lo que disminuye la capacidad de adaptación del árbol.

[10]Si se considera que cada contenedor de un formato alberga un árbol de directorios, consiste en asociar a cada raíz con el nombre de una carpeta en un árbol unificado.

La disposición de los directorios en un sistema unix evidencia eso. Un buen puñado de ellos ha convertido sus nombres en acrónimos retroactivos, y han cambiado de propósito a lo largo del tiempo. En «usr», ya no están los directorios de usuario como al principio, que de hecho ahora se encuentran en «home».

Existen «bin» y «sbin», que están disponibles todo el tiempo en la unidad principal, pero suele haber bajo «usr» otros con el mismo nombre por este motivo. Algunos programas no pueden o no deben ir en «usr»: un claro ejemplo es «mount» (olvidarse el programa de montaje en un formato que aún no ha sido montado suele ser mala idea).

Para hacer esta separación posible, la variable de entorno «PATH» enumera cada directorio donde se almacenan los programas. Los usuarios pueden alterarla para incluir o excluir alguno a lo largo de una sesión. Esta organización permite separar el contenido de «/», «usr» y «home» en tres formatos (p. ej., discos duros).

Si los discos tuviesen *muy* poca capacidad, no todos los programas que acompañan a la distribución cabrían en «bin» y «sbin». Aparentemente, el secuestro de «usr» para el volcado de programas que de otra forma irían en sus directorios habituales, y la posterior escisión de «home» parecen deberse a la falta de espacio (o a la prevención de la misma) en un un disco de poca capacidad.

Hacer de algo así un estándar no es una buena idea. Windows hace posible a través del *registro* la selección del directorio donde instalar cada programa (siempre y cuando los fabricantes de *software* no sean lo bastante sádicos). Idealmente, en unix debería poder escogerse la estructura de directorios adecuada para cada sistema.

Cuando los directorios a utilizar están predeterminados, significa:

1. Que se clasificará a los programas mediante un criterio determinado (*no necesariamente apropiado*, p. ej., que las unidades tienen una capacidad máxima de un par de megabytes).
2. Que no se podrá usar otro criterio.

Incluso si no es así, por cada formato a montar se requiere un directorio. La falta del mismo impedirá realizar añadidos en el árbol unificado, y uno creado con este propósito queda obsoleto en cuanto se deja de usar como punto de montaje.

Así, el problema principal es que los nombres de los directorios indican lo que contienen (no donde se encuentra su contenido), y a pesar de ello se utilizan para indicar donde se debe (o se permite) hacer un punto de montaje.

Estos dos cometidos no están relacionados, y una de sus consecuencias es la creación de directorios a los que no se puede asociar un significado, pero que se añaden para hacer posible un nuevo punto de montaje.

3.2.1 Más o menos jerárquico

El árbol de directorios permite hacer **enlaces fuertes**, es decir, permite que un archivo sea accesible desde varios directorios (otro ejemplo de múltiples identificadores únicos). Quizá incluso en cada directorio utilice un nombre diferente.

Esto es porque, como se ha explicado acerca del espacio de nombres, aquel que recibe cada archivo le es asignado para poder diferenciarlo de los demás y hacerlo accesible. En muchos sistemas de ficheros y formatos, esto significa que el nombre no es parte del archivo, y se suele almacenar dentro de la estructura de datos del directorio. Incluso si solo es accesible a través de uno de ellos, ya está enlazado fuertemente.

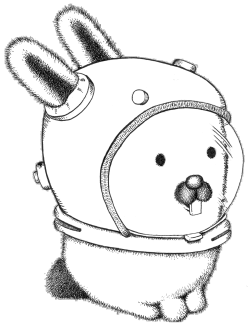
Eso no elimina las ventajas de un sistema realmente jerárquico, porque no se permite enlazar fuertemente (más de una vez) un directorio. Como la jerarquía es a nivel del *espacio de nombres*, que dos nombres «hoja» se asocien al mismo archivo es irrelevante, y realmente no se producen ciclos en la estructura de subespacios. No es posible enlazar fuertemente entre diferentes formatos[11], ni siquiera sería apropiado.

El problema viene con los **enlaces simbólicos** o **enlaces blandos**, que son simplemente archivos comunes con una ruta en su interior. Esto permite hacer referencia a cualquier archivo o directorio, en cualquier formato. Este mecanismo atiende a las limitaciones del enlace fuerte.

En el momento en el que esto entra en juego, y el árbol ya no es jerárquico, y otros problemas aparecen. Los enlaces simbólicos han llegado para quedarse, así que hay que tenerlo en cuenta a la hora de diseñar los algoritmos de los programas, y en especial de los que acompañan al sistema.

[11] Sin importar, en este caso, si son o no formatos de almacenamiento.

3.3 Plan 9 from Bell Labs



Plan 9 es un *sistema distribuido*, actualmente propiedad de Alcatel-Lucent, desarrollado por los laboratorios Bell como sucesor de UNIX desde finales de los 80 hasta 2002. Su primera versión fue lanzada en 1992, y estaba destinada a las universidades. La última versión, la cuarta, es software libre.

3.3.1 Ordenado, simple y osado

Mientras que las otros sistemas resultan inabarcables, este es uno que los alumnos podrían aspirar a leer y modificar de arriba a abajo. Se puede argumentar que esto es porque en comparación con los actuales de clase unix, hace muy poco, pero es justo indicar que también lo hace *muy* bien.

El sistema hace del soporte para múltiples arquitecturas algo casi trivial, el *script* de compilación de un programa medianamente ambicioso (p. ej., el gestor de ventanas) es esencialmente el mismo que el de un «hola mundo», y todo el código fuente acompaña a las distribuciones en un árbol perfectamente ordenado.

Realizar cualquier ajuste es intuitivo (en parte por el énfasis en el uso de texto plano), modificar una aplicación original y reinstalarla es un juego de niños.

Nada de esto impide que sea el primero en soportar UTF-8[12], que los códigos de error y señales dejen paso a cadenas de texto, que sea un claro ejemplo de sistema operativo distribuido accesible, y que proporcione versiones mejoradas tanto de la biblioteca de C como de las aplicaciones del sistema.

[12]No es casualidad, ya que proviene del mismo creador, y surgió durante el desarrollo de la primera versión.

3.3.2 Amigabilidad con el usuario

Al pretender competir contra UNIX en su propio terreno, Plan 9 asume que los usuarios son expertos. En comparación con los actuales sistemas domésticos, *no es amigable con el usuario*.

No obstante, «en el más estricto de los sentidos», los sistemas operativos no tienen usuarios humanos. Conceptos como un planificador de procesos, un gestor de memoria, o una interfaz de llamadas al sistema «amigables con los usuarios» inexpertos son un sinsentido.

El resto proviene de un cascarón que hace de intermediario. Como no es necesario modificar a Plan 9, cambiar esta situación consiste en crear nuevos programas, algo que no entra en conflicto con el funcionamiento interno ni rompe nada esencial. Si este tipo de cuestiones no se han tomado como prioridad, es porque estaba destinado a usuarios expertos desde el primer momento.

Debe tenerse en cuenta que los sistemas pueden tomar medidas para impedir que los usuarios expertos accedan a dichos mecanismos, algo que en este caso hubiese sido inapropiado.

Ambas cuestiones son ortogonales, y por tanto se pueden solucionar al mismo tiempo si se cree oportuno. Al fin y al cabo, un verdadero sistema amigable debiera resultar más productivo incluso para los usuarios expertos.

3.3.3 El sistema de ficheros y 9p

No es necesario tener una parte de la API dedicada al trabajo en red, o a la interacción con los controladores de dispositivo, porque el sistema de ficheros es el que proporciona el punto de entrada a los programas.

Pero esto no se debe a que el sistema de archivos tenga previsto manejar estas funcionalidades *en particular* (alto acoplamiento y baja cohesión), sino que ha sido diseñado para trabajar en una arquitectura cliente/servidor, donde estos últimos son programas en espacio de usuario que proporcionan árboles de archivos (persistentes o sintéticos).

Esto hace que **el sistema de archivos sea extensible a base de crear nuevos programas**.

Es por ello que se necesita un **protocolo del sistema de ficheros**: **9p**[13] sirve a tal propósito. **No es un protocolo de red**. La mayoría de sistemas operativos no necesitan uno de esta clase porque el núcleo incluye al sistema de archivos, y lo único necesario para que haya interacción entre los programas de usuario y este es una llamada al sistema.

Un mensaje en 9p no es un protocolo de red, ni una llamada al sistema, sino un montón de datos que describen una *solicitud de transmisión* o una *respuesta*. Por tanto, es **independiente del medio de transmisión**. Para emplear la red se requiere un protocolo de ese tipo, se puede también trasmitir mediante una tubería, emplear memoria compartida, o cualquier otra opción imaginable.

Además, establece que es *little-endian*[14]. La única consideración que debe tenerse en cuenta si se espera que los mensajes se intercambien entre computadoras es que sea también **independiente de arquitectura**.

Este sistema de ficheros funciona en configuración distribuida, pero tanto la API como el protocolo (*endianness* aparte), aprovechan el cien por cien de su capacidad y diseño incluso en un único equipo, aunque este no tenga previsto conectarse jamás a una red.

Dicho de otro modo, la capacidad de red de Plan 9 combinada con los sistemas de archivos y de seguridad lo convierten en distribuido, aunque las tres cosas no están acopladas entre sí y mantienen su generalidad por si mismas.

Para ilustrar esto, puede emplearse una metáfora en la que un sistema (distribuido o no) es un organismo, y 9p es el oxígeno que este necesita. Para que cada órgano pueda obtener un poco, la sangre (el medio de transmisión) debe ser capaz de transportarlo hasta donde sea necesario. En este caso, es posible que haya más de un medio de transmisión al mismo tiempo, cada uno en un tramo del «circuito».

A modo de introducción rápida, en una comunicación debe existir un servidor de archivos (que contiene un árbol) y un cliente (normalmente el núcleo) a través de una **conexión** bidireccional por la que intercambian mensajes emparejados con una etiqueta. La conexión puede ser compartida por varios clientes, siempre y cuando no haya más de un mensaje con la misma etiqueta simultáneamente (excepto «NOTAG», empleado en «Tversion»).

[13] Concretamente la versión llamada 9p2000.

[14] Para evitar confusiones, recuérdese que significa «desde lo más pequeño hasta el final».

Los números indican el tamaño de variables enteras sin signo. En la siguiente lista, se aprecia que todos empiezan por el *tamaño del mensaje total* y la *etiqueta*. El resto del tiempo, los valores suelen indicar tamaños (excluyendo su propio espacio). Las letras son campos variables que empiezan por un valor de tamaño en 2 *bytes*.

Con todo esto, el nombre de cada mensaje aparece justo tras el tamaño total, así que puede apreciarse que excepto por Rerror, vienen en pares de solicitudes de *transmisión* y sus *réplicas*.

```
size[4] Tversion tag[2] msize[4] version[s]
size[4] Rversion tag[2] msize[4] version[s]

size[4] Tauth tag[2] afid[4] uname[s] aname[s]
size[4] Rauth tag[2] aqid[13]

size[4] Rerror tag[2] ename[s]

size[4] Tflush tag[2] oldtag[2]
size[4] Rflush tag[2]

size[4] Tattach tag[2] fid[4] afid[4] uname[s] aname[s]
size[4] Rattach tag[2] qid[13]

size[4] Twalk tag[2] fid[4] newfid[4] nwname[2] nwname*(wname[s])
size[4] Rwalk tag[2] nwqid[2] nwqid*(wqid[13])

size[4] Topen tag[2] fid[4] mode[1]
size[4] Ropen tag[2] qid[13] iounit[4]

size[4] Tcreate tag[2] fid[4] name[s] perm[4] mode[1]
size[4] Rcreate tag[2] qid[13] iounit[4]

size[4] Tread tag[2] fid[4] offset[8] count[4]
size[4] Rread tag[2] count[4] data[count]

size[4] Twrite tag[2] fid[4] offset[8] count[4] data[count]
size[4] Rwrite tag[2] count[4]

size[4] Tclunk tag[2] fid[4]
size[4] Rclunk tag[2]
```

```
size[4] Tremove tag[2] fid[4]
size[4] Rremove tag[2]

size[4] Tstat tag[2] fid[4]
size[4] Rstat tag[2] stat[n]

size[4] Twstat tag[2] fid[4] stat[n]
size[4] Rwstat tag[2]
```

3.3.4 Por qué emplear un enfoque distribuido

Este trabajo no trata de sistemas operativos distribuidos, sino sobre sistemas de archivos de propósito general. Este sistema de ficheros tendrá todo lo necesario en su extremo para funcionar así. Al proporcionar los otros dos módulos, el sistema operativo que lo utilice adquirirá esas propiedades. Sin embargo, «permitir» tal cosa puede verse como una complicación innecesaria.

No lo es. Es un efecto secundario.

Los sistemas distribuidos se caracterizan porque de cara a los programas, da igual como los equipos colaboren, o si realmente solo hay uno. Siempre funcionan de la misma manera. Pero eso no significa que en un solo equipo tengan alguna ventaja frente a los de propósito general, y efectivamente, utilizar un mecanismo innecesario e inútil puede ser un problema, no muy diferente al que arrastra UNIX desde que entrara en la era de la microcomputadora.

Afortunadamente, no es el caso, porque las consideraciones que habilitan el funcionamiento distribuido en Plan 9 tienen todas un impacto directo ya sea con este o al margen de él. **Las consecuencias que acarrea son el verdadero propósito de este trabajo**.

3.3.5 Vinculación de directorios

Una gran ventaja de trabajar en Plan 9 es que todos los archivos pueden asociarse a cualquier nombre del árbol. En el caso de las carpetas, estas se pueden combinar en forma de pila, así que el punto de montaje «contiene» a todos los archivos de esas carpetas.

El montaje vinculado se realiza mediante el programa «bind» y su llamada al sistema homónima. Un ejemplo es el siguiente:

```
term% lc
bin                                     midiscocuandotengoproblemasdeespacio
term% lc bin
Léeme   cat     echo    ls      mount   mv
term% lc midiscocuandotengoproblemasdeespacio/
Léeme   fs      read    tree
term% cat bin/Léeme
Los programas están aquí, en bin.

Por desgracia, este disco es antiguo ☹.
Aún así no es problema. Al fin y al cabo, esto es Plan 9®.

term% cat midiscocuandotengoproblemasdeespacio/Léeme
Este disco almacena algunos programas.
(Ver nota en bin).
```

Hay dos directorios, «bin» y el punto de montaje de un disco. Como puede comprobarse, aunque ambos contienen un archivo llamado «Léeme», no son el mismo archivo.

Es hora de aplicar un pequeño cambio.

```
term% bind -b midiscocuandotengoproblemasdeespacio/ bin
term% lc bin && cat bin/Léeme
Léeme   Léeme   cat     echo    fs      ls      mount   mv      read    tree
Este disco almacena algunos programas.
(Ver nota en bin).
```

Lo primero, es que todos los archivos que residían en el disco ahora son visibles en «bin», junto con los que ya había. Lo segundo, pero también lo más sorprendente, **es que hay dos archivos con el mismo nombre**. ¿No eran identificadores únicos?

Y siguen siéndolo. El disco se ha apilado **sobre** «bin», por lo que cualquier intento de interacción siempre hará referencia al archivo «Léeme» contenido en este. El funcionamiento es exactamente el mismo al de los identificadores de las variables en cualquier lenguaje de alto nivel común, donde las variables que están en un ámbito interior ocultan a los identificadores que provienen de uno exterior.

El parámetro *before* evita que «bind» utilice su comportamiento por defecto, el de reemplazo (similar a un «mount» de unix común, donde todo el contenido de la carpeta queda oculto durante el montaje). De hecho, también pueden hacerse vínculos con aquellos archivos que no sean carpetas.

Por supuesto, existe el parámetro *after*:

```
term% unmount bin
term% bind -a midiscocuandotengoproblemasdeespacio/ bin
term% lc bin && cat bin/Léeme
Léeme   Léeme   cat     echo    fs      ls      mount   mv      read    tree
Los programas están aquí, en bin.

Por desgracia, este disco es antiguo ☹.
Aún así no es problema. Al fin y al cabo, esto es Plan 9®.
```

Ahora, el archivo Léeme asociado al mismo nombre en «bin» es el archivo original. Lo mismo pasará con el resto de programas: los situados en «bin» no serán sustituídos por los que aparezcan en el otro directorio.

Aún hay un problema más que debe resolverse. ¿Y si se desea **crear** un nuevo archivo, en que carpeta acaba? Durante el montaje, los directorios pueden contar (o no) con el parámetro *create* («-c»). Como resultado, el primero de la pila con esta opción será el lugar adecuado.

Este mecanismo puede usarse en lugar de los enlaces simbólicos, pero está claro que permite algunas cosas que estos no pueden hacer.

3.3.6 Múltiples espacios de nombres

Otra de las ventajas es que aunque se manipule el espacio de nombres, no todos los programas se verán afectados. Los procesos, al bifurcarse, pueden decidir pertenecer a una copia del espacio actual, o crear uno en blanco. De esa forma, el y sus colaboradores pueden reconfigurar el árbol de directorios por completo para atender una necesidad concreta.

Por ejemplo, los *scripts* de inicio de cada usuario realizan un «bind» de su directorio local de programas, «/usr/<nombre>/bin», sobre el directorio del sistema. De esta forma, cada uno pude disponer de una colección diferente de programas, e incluso instalarlos localmente sin molestar a nadie.

3.3.7 Haciendo a punto-punto bien[5]

El núcleo de Plan 9 lleva la cuenta de la ruta seguida para cada archivo abierto. Esto junto con la tabla de montajes ayuda a solucionar un pequeño problema de los enlaces simbólicos: no hay forma de saber como se llegó a un lugar concreto del árbol, por lo que no hay garantía de regresar al «directorio padre» esperado al utilizar «..».

La solución en este caso es muy simple, porque solo hay definir el comportamiento de «..» como la eliminación del nombre del directorio anterior.

3.3.8 Sin «setuid» ni «setgid»

Distribuído o no, manipular del espacio de nombres puede tener severas consecuencias, porque si se vincula un archivo con estos bits activados en el nombre de otro, se puede hacer que un programa que espera utilizarlo abra la caja de Pandora.

La protección ante esta pequeña desventaja también es bastante simple, y es evitar esta característica y buscar otra manera menos peligrosa de hacer lo mismo.

3.3.9 Sin usuario «root»

Por motivos similares, no hay superusuarios en Plan 9. En un entorno distribuído, el concepto es absurdo. ¿Y fuera de él? Algunas distribuciones de sistemas operativos actuales[15] han decidido que es tremendamente peligroso, y han decidido deshacerse del superusuario.

En su lugar, cada recurso está bajo el control de algún usuario, y por tanto le corresponde a él la seguridad del mismo.

3.3.10 El legado de Plan 9

Puede apreciarse como algunas características de los sistemas operativos actuales han tomado a Plan 9 como inspiración. Algunos ejemplos son:

- La codificación UTF-8, muy popular en el mundo unix.
- El uso contemporáneo del directorio **/proc**.
- La llamada al sistema «rfork» inspira a la de FreeBSD[16] y a «clone» en Linux[17].
- El comportamiento de «mount» junto con el *flag* «--bind» en Linux.
- Los espacios de nombres en Linux.

Algunas de sus herramientas también han sido bifurcadas, adaptadas, o han servido de inspiración. Durante el desarrollo de este trabajo fue particularmente útil usar algunas de las más emblemáticas.

[15] GoboLinux, por ejemplo.
[16] Esto es mencionado en el manual, en la página RFORK(2).
[17] Según «El Arte de la Programación en Unix».

3.3.11 Derivados de Plan 9

Algunos sistemas operativos están, «genéticamente»[18] o no, conectados a Plan 9.

Es destacable Inferno, iniciado también en Bell, y que además de poder ejecutarse directamente «sobre el metal» puede funcionar como una aplicación de usuario sobre múltiples sistemas operativos (incluso de algunos Windows de la rama NT).

Mientras tanto, usuarios de todo el mundo no han perdido el tiempo:

- 9front es una distribución actual y muy activa, con toda una comunidad detrás.
- Akaros, el sistema operativo de la *Universidad Californiana de Berkeley* (UCB), y que ha motivado el cambio de licencia.
- Harvey, un sistema modernizado de 64 bits, que pretende ser compatible con POSIX y sus programas.
- Clive[19], influenciado por este y Nix[20], detrás del cual está Francisco J. Ballesteros.

Y otros. Algunos proyectos como Glendix han desaparecido con el tiempo, pero muchos se mantienen activos con el humor y las ganas de siempre.

3.4 Plan 9 from User Space

También conocido como «plan9port», puede dividirse en tres partes para los propósitos de este trabajo.

La primera es una colección de programas adaptados desde su entorno nativo al de unix. El editor Acme, Mk, y un dúo compilador/cargador de estilo Plan 9 (9c y 9l)[21] se incluyen en ella.

[18] Es decir, a nivel de código fuente.

[19] El nombre del mismo y el del protocolo del sistema de ficheros son una clara referencia al mítico computador, el Sinclair ZX Spectrum.

[20] No confundir con NixOS, una distribución de Linux.

[21] Un vistazo detallado revela que en realidad son *scripts*.

La segunda parte es una colección de bibliotecas. Entre ellas aparece la biblioteca de C de Plan 9, pero también otras de gran utilidad que acompañan al sistema, incluso algunas nuevas como ayuda. No hay ningún motivo para reescribir lo que se ha adaptado ya, al contrario, algunas de estas bibliotecas son dependencias de los programas de este trabajo.

La tercera parte es la propia API de Plan 9 y su implementación, que en realidad forma parte de los programas y bibliotecas que se han mencionado. Esta API es implementada mayormente como un envoltorio de la API unix. Es la última pieza que permite desarrollar aplicaciones en gran medida «portables» entre ambos sistemas, algo así como APE[22] para Plan 9.

El presente trabajo es complementario a esta tercera parte. Dado que es un envoltorio pensado por y para unix, la funcionalidad de «plan9port» depende de la de este, y por tanto no puede hacer todo de lo que Plan 9 es capaz. En cambio el objetivo del trabajo es implementar el sistema de archivos completo, incluso si en ultima instancia este se sostiene en el de unix. El propósito de esto es que, algún día, funcione en un núcleo propio y esta dependencia sea eliminada.

3.5 Alternativas a Plan 9 como punto de partida

Por un lado «NFS», dado que como 9P también es independiente de arquitectura y del protocolo de transmisión, además de ser bastante simple. No hay gran diferencia en este sentido, pero hay algunas cosas a tener en cuenta.

La primera (y seguramente la más importante) es que 9P ha sido pensado como el protocolo interno de Plan 9, y por tanto encaja perfectamente con sus características.

La segunda es que ha sido pensado para utilizarse como el lenguaje nativo del sistema de archivos (por lo que no es necesario algo como «VFS»). Esta característica es muy importante en este caso, por lo ya explicado. Por supuesto podría intentarse algo así con NFS, pero hay un motivo más.

La tercera razón para escoger Plan 9 y su protocolo es que además ya cuenta con excelentes ejemplos bien probados de como usar el protocolo a nivel interno.

[22] Entorno ANSI/POSIX. Curiosa elección de siglas.

Por otro lado, para el propósito deseado, un sistema de tipo Plan 9 puede competir con sistemas como Minix 3, Linux o BSD. Por desgracia, las personas a cargo del primero están trabajando por una mayor compatibilidad con NetBSD, y muchos programas que estaban disponibles en versiones anteriores de Minix 3 han dejado de funcionar. En cuanto a los otros dos, son proyectos increíblemente inmensos. De hecho, el código de Linux es muy inestable (es decir, que cambia con rapidez), lo que le permite estar a la última, pero lo vuelve poco apropiado en este caso en particular.

Capítulo 4

Entorno y utensilios

4.1 Afinidad entre UNIX y Plan 9

UNIX es en si mismo un entorno de desarrollo. Sin embargo, no es un entorno integrado, y eso es visto, sin duda, como una ventaja por parte de sus usuarios.

Los programas que lo componen realizan idealmente una sola tarea, en la que se especializan. En cambio, los IDE reúnen la mayor parte de las herramientas necesarias en un solo programa, lo que permite que estas compartan sus funcionalidades y se repartan el trabajo fácilmente.

La diferencia es que los programas de UNIX son capaces trabajar juntos para sumar sus funcionalidades y de repartirse el trabajo a través del propio sistema operativo, pero sin todo ese acoplamiento.

Esto hace que sea más fácil de mantener, que pueda incorporar nuevas funcionalidades (programas), y que estos se puedan reemplazar sin cambiar una sola línea de código (instalar). Cada pedazo se puede tratar independientemente, pero aporta su funcionalidad a todo el sistema. Para usar unix, ni siquiera es necesario cargar todos los programas que lo componen, solo los que se necesitan.

Los programas hechos en un unix determinado son fáciles de reconstruir en otro (con un poco de suerte sin cambios), además hay soporte para literalmente cientos de arquitecturas. Vale la pena señalar que muchas de las características que algunas personas atribuyen exclusivamente a los IDE (si no todas) pueden lograrse combinando herramientas independientes, todo de depende de escoger las adecuadas.

Aunque Plan 9 no es unix, aún puede considerarse descendiente de UNIX. Lo primero puede desorientar a los usuarios que esperan lo contrario, pero lo segundo significa que hay muchas similitudes esenciales y una misma herencia, que a veces se manifiesta de maneras inesperadas.

4.2 Debian 8 (Jessie)[1]

Debian es una distribución de Linux. Lo bueno de Linux es que hay muchas distribuciones libres, gratuitas y fáciles de conseguir.

Para este trabajo se pudo utilizar cualquier otra distribución. Es más, se escogió e instaló esta en particular a tal fin. Incluso se anima a cualquiera que quiera reutilizar los resultados a usar otra diferente si así lo desea, no sin advertir que algunas rutas y configuraciones podrían cambiar y dicha elección podría acarrear un poco más de trabajo. Dicho esto, se confía en que muy poco.

Debian implementa un «GNU con Linux» , algo a tener en cuenta dado que se usarán los programas de GNU. Por curiosidad, también se ha usado GNOME 3.

4.3 Plan 9, 4ª edición

Se puede descargar una imagen del CD de instalación de esta distribución. Se aconseja habituarse a sus herramientas y leer los documentos de introducción, que se despliegan al iniciar sesión en la configuración por defecto.

Instalar Plan 9 requiere prestar atención. Al iniciar se recomienda elegir una resolución de 1280x768x24 o de 1600x1200x32 y elegir el modo «VESA». La instalación puede transcurrir en modo ”solo texto” si no se elige correctamente una resolución (o por algún otro motivo más difícil de controlar), normalmente eso supone que el sistema instalado tampoco podrá iniciar Rio[2] por lo que puede ser buena idea reiniciar para encontrar una resolución soportada y apropiada. Si ninguna funciona, la resolución por defecto en el modo por defecto debería ser suficiente para «el primer vuelo».

[1] Las versiones de Debian reciben su nombre de personajes de Toy Story. El nombre de la distribución es la fusión del nombre del creador y su ex-mujer.

[2] El sistema de ventanas de Plan 9, de Rob Pike.

Si más addelante se desea buscar nuevas resoluciones, la orden *aux/vga -m vesa -p* muestra las resoluciones del modo vesa, y *aux/vga -m vesa -l <resolución>* la cambia en un sistema ya iniciado, para hacer los cambios permanentes hay que modificar el archivo */n/9fat/plan9.ini*, aunque ese archivo no aparece hasta que no se invoca *9fat:*, (con los dos puntos). No hay que olvidar desmontar con *unmount* una vez se termine.

Además, hay mas modos que se pueden probar, o incluso añadir nuevas entradas a la base de datos.

Para instalaciones no virtuales, puede ser apropiado utilizar la distribución «9front» en lugar de la de Lucent, ya que tiene más controladores y en general mejor soporte para hardware «moderno». Los pasos a seguir en la instalación son diferentes, las instrucciones pueden verse en su página web.

Una vez puesto en funcionamiento, leer los *papers* y el manual (volumen 1[6])[3] es más que aconsejable. Imprescindible el libro de Francisco J. Ballesteros[7] durante el aprendizaje.

4.4 VirtualBox

Aunque no se recomienda su uso porque aparecen incompatibilidades entre versiones a menudo, es lo que se ha utilizado por motivos prácticos. Las dos distribuciones (Debian y Plan 9) se han utilizado en máquina virtual, y VirtualBox era muy fácil de obtener en el anfitrión (y en otros con la misma distro que se iban a usar también).

En cualquier caso es preferente el uso de máquinas virtuales para este tipo de cuestiones, porque instalar en un equipo físico es menos interesante cuando se van a realizar pruebas que pueden «romper» el sistema cada dos semanas. Además se pueden transportar las máquinas a cualquier lugar, se pueden instalar varias a la vez con diferentes configuraciones, y así instalar ambas distribuciones con arranque múltiple en la misma máquina no es materia obligada (e incluso mala idea).

[3] Por desgracia la página cae con frecuencia. No obstante, todo se puede encontrar en el propio sistema (o desde algún repositorio).

4.5 Configuración del huésped[4]

En primer lugar se instalaron las «Guest Additions» insertando la imagen («Host + D»), dirigiéndose al directorio donde se encontraba la imagen montada, adquiriendo privilegios elevados (orden «su»), y ejecutando el *script* «VBoxLinuxAdditions.sh».

Evitar la búsqueda de actualizaciones desde el CD de instalación es conveniente. Mediante privilegios elevados (antes conviene desactivar los *backups* de Gedit) hay que modificar /etc/apt/sources.list añadiendo el símbolo «#» ante las líneas relacionadas con el CD.

El *toolchain* de GCC debe estar instalado («apt-get install gcc»). También deben instalarse algunos paquetes relacionados con X11 antes de hacer lo propio con plan9port («apt-get install libx11-dev libxext-dev»). Por si surge algún problema con este tipo de cuestiones, «apt-file» puede ser un gran aliado.

El esquema de colores en Gedit[5] y la terminal se cambió por el estilo «solarizado» de Plan 9 (no tanto por preferencia, sino para evitar cansancio visual debido al contraste entre las pantallas). Además, se cambió la fuente del editor por una proporcional. Es más fácil no fiarse de las columnas cuando el editor no invita a ello (además, estas suelen ser más agradables). Con la selección de *plugins* extra, el dibujado de espacios y la integración con Git resultan muy útiles.

A propósito de «Git», es otro programa a instalar. Por comodidad, se aconseja también otros como «Vim», «cloc», o «cscope».

Añadir un atajo para abrir la terminal y el intérprete también puede ser útil («configuración, teclado, atajos, combinación personal», el programa es «gnome-terminal»), por ejemplo «Ctrl + Alt + T».

[4] La instalación de «plan9port» se explica junto con el desarrollo del trabajo[6], aunque algunos de los pasos explicados aquí son necesarios para satisfacer las dependencias de dicha instalación, por lo que deben realizarse previamente.

[5] Se utilizó el tema libre de Craig Russell, que se puede conseguir desde su repositorio en Git.

Recomendaciones específicas del entorno de Gnome son habilitar el clic intermedio para enviar las ventanas al fondo en Gnome Shell y cambiar el comportamiento de Nautilus para abrir los archivos con un solo clic. Para lo primero se puede emplear «gnome-tweak-tool». En la sección «ventanas» y entonces en «acciones de la barra de título» puede encontrarse la opción apropiada. Lo segundo se consigue desde el menú de preferencias del propio explorador de archivos.

Capítulo 5

Dentro de Plan 9

5.1 Introducción a las utilidades

Algunos de los programas más destacables son los siguientes.

5.1.1 Los editores de texto

Sam

Ed es un anteriormente conocido editor de texto para UNIX en línea de órdenes, con un conjunto reducido de mandatos. Sirvió de inspiración a Vi, que a su vez fue sucedido por Vim[1], y con cada descendiente, una cantidad ingente de nuevos mandatos se añadieron al léxico de los programadores.

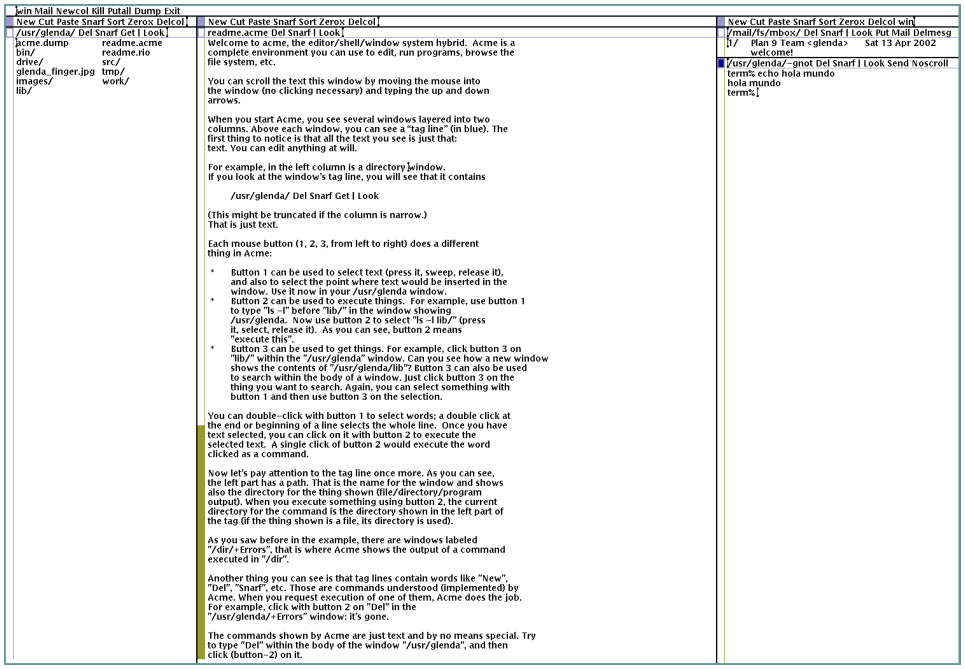
Vi/Vim, de forma similar a Emacs, se ha ganado un cierto aire de misticismo. Se cuenta que solo los usuarios expertos son capaces de dominar cada una de sus funcionalidades, en cuyo caso tienen en sus manos un editor mucho más poderoso y rápido, en el que el ratón es una perdida de tiempo ya que todo está a una pulsación de distancia en el teclado.

[1] Vi Improved

Se podría pensar que «la gente de UNIX» tiene muy buena opinión acerca de los todopoderosos mandatos de Vi, pero la realidad parece ser diferente. Rob Pike apreciaba la sencillez de Ed, y desarrolló Sam, que *«solo agrega una docena de nuevas ordenes»*[2]. Curiosamente, Sam se ejecuta en modo gráfico, otorgando un método de entrada bidimensional del que la CLI y el teclado no disponen, en favor del ratón.

Hay quien apoya que, inesperadamente, el ratón es el más rápido para muchas tareas[8] mientras que la percepción humana juega una mala pasada a los observadores haciendo parecer lo contrario, por lo que los editores de Plan 9 se basan en una filosofía opuesta a la de Vi/Vim. A nivel de interfaz (¡incluso en el gestor de ventanas!), los botones direccionales «arriba» y «abajo» son ocupados para el desplazamiento vertical de la vista sobre el contenido, y no del cursor, con lo que es necesario usar el ratón para realizar tales movimientos.

Acme



Acme sucede a Sam, pero ya no es solo un editor de texto, sino también una terminal, un sistema de ventanas, y muchas cosas más.[3]

[2] Eric S. Raymond es un conocido hacker asociado al movimiento Open Source.

[3] El nombre hace referencia a su carácter polifacético, esto es, su capacidad para hacer de todo.

Sam usa la interfaz gráfica para superar una limitación de control y poco más, pero Acme introduce nuevas funcionalidades con ella en mente. Acme no emplea complejos *scripts* para ser configurado como sus primos lejanos, y ni siquiera se puede cambiar su tema de colores, pero a la hora de utilizarlo es donde puede notarse su flexibilidad, otro cambio de mentalidad con respecto a Vi. En lugar de utilizar *plugins*, Acme es «extendido» trabajando con los programas y *scripts* instalados en el sistema.

Conviene aprender a invocar y lanzar órdenes, crear una ventana con una instancia del intérprete a modo de terminal («win»), buscar palabras en el texto, abrir archivos, y copiar o pegar (utilizando las combinaciones de botones del ratón «1 + 2» y «1 + 3»).

Nótese que las barras de desplazamiento en Acme se comportan como en Rio, pero en implementación propia. El funcionamiento es similar al de las barras de «Athena»: manteniendo botón «2» se puede mover la barra, mientras que pulsaciones del «1» o del «3» permiten subir o bajar respectivamente. La medida en la que se desplaza la barra en este caso depende de la posición donde se pulse, arriba los saltos son muy pequeños (¡sin importar si se desea subir o bajar!). Concretamente, es proporcional al número de líneas.

El propósito de este comportamiento es que al hacer clic con el botón «3» en una línea, esta se convierta en la primera visible. En caso de equivocación, una pulsación del botón «1» devuelve al usuario a la posición anterior.

Curiosamente, el tamaño del tirador es proporcional al texto que se encuentra en la ventana en ese momento.

5.1.2 Mk y Rc

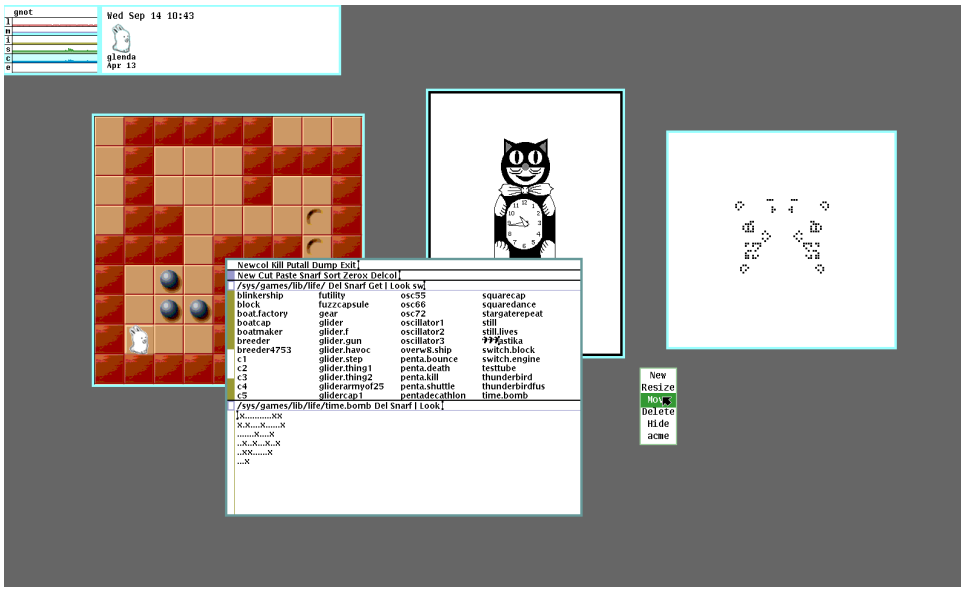
Mk y Rc son programas originarios de *UNIX version 10*[4], y son característicos de Plan 9. Lo primero es familiarizarse con las diferencias.

Mk es considerado como una mejora sobre Make. Pero más allá de eso, es importante saber que Plan 9 cuenta con algunos «mkfiles» prototipo que permiten abordar diferentes situaciones. Estos no deben copiarse, sino incluirse apropiadamente en el *mkfile* de cada programa, lo que facilita mucho la creación, ampliación y compilación cruzada de los mismos.

[4] Versión que nunca fue lanzada públicamente, aunque circulan fragmentos de su manual por la web.

En cuanto a Rc, es una *shell* que pretende ser simple y eficaz. Algunos usuarios encontrarán con sorpresa que no tiene autocompletado ni historial. El primero está incorporado en Rio, y puede desencadenarse con la combinacion de teclas «Ctrl + F». En cuanto al segundo, se decidió no incluirlo, aunque existe algún truco escudriñando en el *buffer* de la ventana....

5.1.3 Rio, el gestor de ventanas



Rio reemplaza a 8½[5], que a su vez está inspirado en los sistemas de ventanas de UNIX, como mpx y mux[6]. De hecho, toda la interfaz utiliza copiosamente los tres botones del ratón y menús desplegables, pero no hay barras de título ni casi otras (así llamadas) «decoraciones»[7]. Las opciones intentan ser reducidas pero flexibles, y las ventanas no deciden por si mismas que espacio ocupar dejando esa la decisión al usuario.

Pero hay algunas decisiones curiosas en su diseño. En primer lugar, Rio es un **servidor de ficheros**. Mediante las operaciones de manipulación del espacio de nombres, el programa que se ejecuta en cada ventana tiene a su disposición unos archivos dedicados en rutas predeterminadas. Es decir, misma ruta, diferente archivo.

[5] (¿Tal vez otra referencia cinematográfica?).
[6] No es casualidad, porque también los hizo Rob Pike.
[7] A excepción de las que vienen del «lado del cliente».

Al escribir dentro de los archivos del árbol de Rio, este interpreta cada interacción como una orden, pudiendo mover a las ventanas, cambiar su tamaño, tomar el control del ratón y otras muchas cosas. Los programas pueden invocar las mismas órdenes que proporcionan los menús a través de esta interfaz.

Como se ha explicado, el que emplea la biblioteca de autocompletado es Rio. Esto es posible porque además de contener el código de las ventanas (algo que entra dentro de lo que se puede esperar), también incorpora el de las terminales gráficas. Lo que es más, las ventanas son las terminales.

¿Y si se va a hacer uso de las capacidades gráficas, como controlar el ratón o dibujar una imagen? Al crear una nueva ventana, esta actúa como terminal, pero a medida que el programa que hospeda intenta tomar el control, la ventana deja de ocuparse de ciertas cosas, hasta no parecer una terminal en absoluto.

En el modo por defecto, aprovechando que la ventana conoce el directorio de trabajo del programa hospedado, ofrece el autocompletado no solo a la *shell* sino a cualquier programa, pero con un par de pequeños inconvenientes. El primero, es que lo único que puede rellenar es el nombre de los archivos. El segundo (bastante más grave) es que la ventana vive en un **espacio de nombres separado**.

Imagínese lo que podría ocurrir al montar un disco[8] desde la *shell*. ¡Esta manipulación del espacio de nombres pasa inadvertida a Rio! Si el punto de montaje es, por ejemplo, un directorio vacío, el autocompletado no funcionará.

Pero si no lo estuviese, las consecuencias son mucho peores. La ventana empezará a sugerir nombres de archivos que ni siquiera se encuentran en esa localización.

A pesar de estos obstáculos, Rio y Acme ofrecen una experiencia de usuario diferente a lo que se suele encontrar en la mayoría de distribuciones de otros sistemas operativos.

Como todo lo demás, los botones «2» y «3» se usan con frecuencia, en este caso muestran diferentes menús. Las primeras cosas a aprender son copiar y pegar, y activar el desplazamiento (*scroll*) automático.

[8]Realmente, un proceso servidor de ficheros leyendo y escribiendo en el formato de un disco.

5.2 Directorios, órdenes y consejos básicos

Una vez instalado, puede observarse que el código fuente de la distribución es hospedado en el árbol de directorios, en su mayoría bajo el directorio /sys/src. Y no solo lo relativo al núcleo, también a la interfaz gráfica, los servidores de ficheros, las aplicaciones, etc...

Si se desea alterar el comportamiento de las barras de desplazamiento, o los colores del editor de texto, la mayoría de las veces basta con copiar el directorio correspondiente, realizar cambios en el código al gusto del usuario y emplear la orden **mk install**, lo que reemplazará el componente ya instalado por el nuevo. Si se desea volver atrás, puede utilizarse la misma orden desde otra copia del directorio original.

Una de las primeras cosas a tener en cuenta es que los programas para CLI del sistema son diferentes a los de BSD o GNU[9] (y en general, significativamente más simples).

Para realizar la tarea anterior, por ejemplo, el programa de copia («cp») no tiene intencionadamente opción recursiva[10]. En su lugar, el *script* «dircp» permite llevar a cabo esa tarea.

La organización del árbol de directorios es realmente intuitiva. Los programas comunes se encuentran en /sys/src/cmd, mientras que la biblioteca de C de Plan 9 se sitúa en /sys/src/libc, algo a recordar porque las **llamadas al sistema** están declaradas en ella.

En realidad, no todas se implementan en forma de **llamadas al núcleo** (contenidas en /sys/src/libc/9syscall), sino que algunas llamadas al sistema *envuelven* a otras, y por tanto son implementadas por la biblioteca. En su mayoría estas se reúnen en /sys/src/libc/9sys[11].

[9] Especialmente en este último, los argumentos de tipo *flag* pueden situarse en cualquier lugar, mientras que en Plan 9 deben situarse antes que los de otro tipo.

[10] Probablemente esto se debe a que implementar la recursividad en cada programa ha sido considerado una duplicación de código innecesaria e indeseable.

[11] Nótese que a las aplicaciones no les importa que llamadas al sistema son también llamadas al núcleo, pueden serlo todas o ninguna. Cualquier reimplementación de Plan 9 con la misma *API* del sistema puede decidir cuales pertenecen a cada categoría y mantener la compatibilidad a nivel de código fuente.

Las llamadas al núcleo se componen de una parte en espacio de usuario, que solo utiliza una instrucción especial dependiente de arquitectura y cuyo cometido es devolver el control al núcleo para que ejecute un subprograma correspondiente. La otra parte es dicho subprograma en espacio del núcleo, que es la verdadera implementación de la llamada. Para encontrarla, basta con buscar en /sys/src/libc/9 subprogramas con los mismos nombres que los de la biblioteca, a excepción del prefijo «sys» que los acompaña.

5.3 El lenguaje C en Plan 9

Plan 9 está escrito en su mayor parte en lenguaje C, como no podía ser de otra manera. Sin embargo, no es el lenguaje C estándar, sino un dialecto del mismo creado para Plan 9.

Esto no es tanto una desventaja para la compatibilidad como pueda parecer[12], especialmente cuando la API de unix es mucho más determinante.

Plan 9 cuenta también con una biblioteca de C fuera del estándar, y junto con las bibliotecas del sistema forman una mezcla curiosa.

5.4 Maravillas de la compilación cruzada

El primer programa escrito para Plan 9 fue su compilador. Pero en realidad no tiene uno, sino al menos una decena: uno por arquitectura (Intel x86, Motorola, SPARC, MIPS, ARM, Alpha y variantes).

Cada compilador, cargador[13] o ensamblador tiene un nombre diferente, la mayoría formados por un número y la letra «c». No se utiliza un *frontend* para invocar automáticamente al cargador correspondiente (cuyo nombre acabará en «l»).

Por otro lado, Plan 9 tiene directorios tales como */386*, */amd64*, */mips* o */sparc*, donde residen algunos *fuentes* y *scripts* dedicados a cada arquitectura.

[12] En cualquier caso el Entorno ANSI/POSIX, o *APE* (curiosa elección de siglas) ha sido desarrollado para este tipo de escenarios.

[13] Así es como se llaman los enlazadores en Plan 9.

Todo esto es aprovechado por **Mk**, con la consecuencia de que reconstruir un programa de Plan 9 desde una máquina de cierta arquitectura para otra diferente sea pan comido, hasta el punto de no ser necesario modificar el *script* (**mkfile**) de un programa para lograrlo, y es suficiente con ajustar una variable en el intérprete.

En resumen, en Plan 9 la compilación cruzada es una característica predeterminada.

Capítulo 6

Diseño e implementación

Antes de nada, es necesario instalar «plan9port». Esto es demasiado importante como para tratarlo por separado.[1]

Desde https://swtch.com/plan9port/ puede encontrarse un enlace para descargar un *tarball* con el código fuente necesario para poderlo instalar en un sistema unix. Las instrucciones de instalación pueden encontrarse en INSTALL(1).

Aunque puede realizarse en dos partes, en resumen, un buen sitio para albergar el código es /home/<usuario>/local/plan9port (o *.local* si se prefiere). Después solo hay que ejecutar el *script* de instalación y esperar. En las máquinas más lentas puede llevar entre media hora y una hora.

./INSTALL

Al terminar, se solicita al usuario que ajuste la variable de entorno PLAN9 apropiadamente. Para ello, se puede editar el *script* .bashrc (o el equivalente del intérprete escogido). En este caso se ha utilizado el siguiente ajuste.

[1] Hay algunas dependencias que cumplir. Es posible que haya que realizar algunos pasos previos, en este caso particular detallados en la sección dedicada a la configuración del huésped[4.5].

```
if [ -z ${PLAN9+x} ] && [ -d ~/local/plan9port ]; then
        PLAN9=~/local/plan9port
        export PLAN9
        PATH=$PATH:$PLAN9/bin
        export PATH
fi

if [ -n $PLAN9 ]; then

        echo "Plan 9 from User Space environment"
else
        echo "PATH not set for Plan 9 from User Space usage"
fi
```

Este sistema no vuelve a concatenar PLAN9 a PATH si se abrieran recursivamente varios intérpretes, aunque tampoco sería un impedimento que así fuese. Un procedimiento similar permite situar algunos programas de prueba en la carpeta del usuario.

Lo bueno de este sistema es que, además de hacer que los programas y *scripts* de «plan9port» (tales como Acme, Mk, lc, sig, o lookman), al añadir un directorio de binarios a PATH también se puede acceder a las páginas del directorio del manual que se encuentre junto al primero. En este caso se sigue la nomenclatura de unix y esas páginas se han sido movidas a las secciones correspondientes. La sección 9P es especial, porque incluye todo lo relativo a dicho protocolo.

6.1 El truco de Plan 9 from User Space

Los núcleos *unix-like* no tienen las llamadas al sistema necesarias, ni tampoco realizan las mismas tareas. En su lugar, «plan9port» utiliza el sistema de archivos del núcleo correspondiente. Para suplir algunas de sus carencias realiza un par de maniobras ingeniosas.

En primer lugar, los servidores de archivos 9P no pueden conectarse directamente al sistema de archivos. Uno de los mecanismos que «plan9port» puede utilizar es **FUSE**[2], siendo el caso de Linux. Para ello utiliza un programa llamado «9pfuse»[3].

[2] Sistema de archivos en espacio de usuario a través de un módulo situado en el núcleo.
[3] Cuyo código está situado, por supuesto, en $PLAN9/src/cmd.

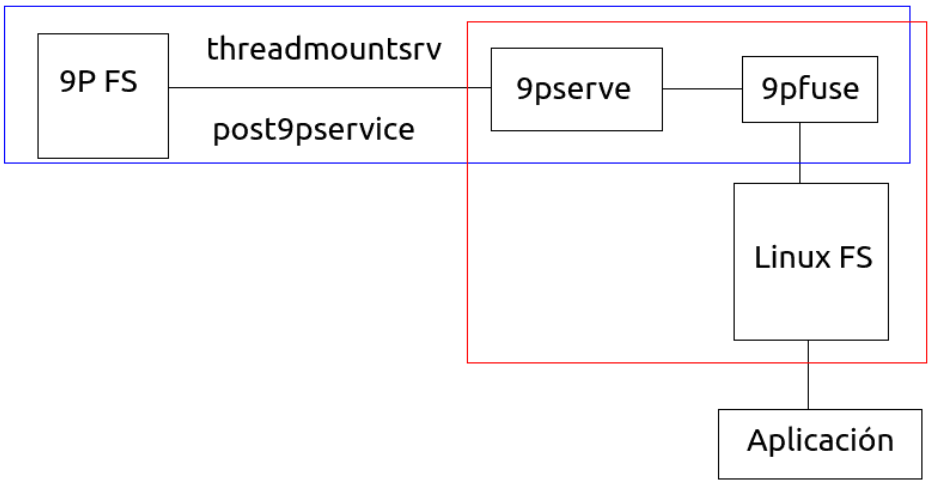
Este programa a su vez se conecta a «**9pserve**», que es capaz de mantener varias conversaciones en 9P y multiplexarlas en una única en el extremo opuesto (9PSERVE(4)). Cada servidor utiliza a «9pserve» como pasarela.

Al iniciar el proceso correspondiente, este se bifurca para crear un proceso «9pserve» (binomio **rfork/execl**). A su vez, «9pserve» puede generar un proceso «9pfuse», de tal manera que cada proceso servidor cuenta con su propio multiplexor.

Para que los tipos y llamadas de Plan 9 respondan correctamente, en las cabeceras (excepto cuando se utiliza una macro especial para impedirlo, **NOPLAN9DEFINES**) de definen las macros correspondientes para expandirse como funciones del mismo nombre con el prefijo «p9», y así evitar usar la llamada del sistema anfitrión directamente.

Hay que leer la documentación con cuidado, pues no todo funciona igual y algunas cosas del sistema original carecen de implementación o funcionan de manera inesperada.

Los servidores no se bifurcan directamente, sino que utilizan **threadpostmountsrv**[4] en la biblioteca «**lib9p**»; o de forma más directa, **post9pservice** en la biblioteca «**lib9**» (que básicamente es el equivalente a «libc» en Plan 9).



[4]Hay una contraparte fuera de la biblioteca de hilos, **postmountsrv**, y aunque no ha surgido la ocasión de probarla, su implementación es muy diferente a lo que se puede esperar y no parece ser utilizada por los otros programas.

De esta forma, por lo que respecta al núcleo, *cada* servidor de archivos es un conjunto de tres procesos: el servidor 9P, «9pserve» y «9pfuse», mientras que por lo que respecta al servidor 9P y según el protocolo, «9pserve» y «9pfuse» son el cliente, en Plan 9 interpretado simplemente por el núcleo.

Resulta que estos dos programas se conectan a través de un **Socket en Dominio de Unix**[5], que por casualidad fueron escogidos desde el principio para implementar este proyecto.

En «plan9port», todo fragmento de la biblioteca principal relacionado con «dial» (DIAL(3)) utiliza esta interfaz[6]. Se puede incluso solicitar a «9pserve» que se comunique con el servidor que lo bifurcó a través de uno, de la misma manera que con los clientes.

A falta de espacios de nombres y del mecanismo para publicar el descriptor de archivos nativo, los *sockets* aparecen en un directorio temporal en rutas del estilo «/tmp/ns.'$USER"$DISPLAY'/<SERVICIO>»[7].

6.2 Mecanismo de comunicación interproceso

A diferencia de «plan9port», se necesitaba una forma de reemplazar las llamadas al sistema por completo, y además, sin modificar el núcleo. Para ello, se necesitaba un mecanismo con el que establecer una comunicación entre el servidor de archivos central y las aplicaciones.

Se escogieron los **Sockets en Dominio de Unix** (*UDS*) a tal fin por ser los más adecuados. Algunas de las alternativas más prometedoras eran las tuberías con nombre (las tuberías comunes exigen que el servidor se bifurque para crear a las aplicaciones) y los buzones de mensajes.

Entre las restricciones, el servidor se debía poder comunicar con múltiples procesos (a los que debía diferenciar de alguna forma), en comunicación bidireccional, y con mensajes de tamaños variables. Por todo ello, los *sockets* resultaron la opción más flexible y escalable.

[5] Y dicho sea, la biblioteca de FUSE también los utiliza[9].

[6] En modo «stream», que además tienen la ventaja de ser bidireccionales (*full duplex*).

[7] Existe la posibilidad de que la variable de entorno «DISPLAY» esté estructurada de diferente manera dependiendo del sistema unix concreto.

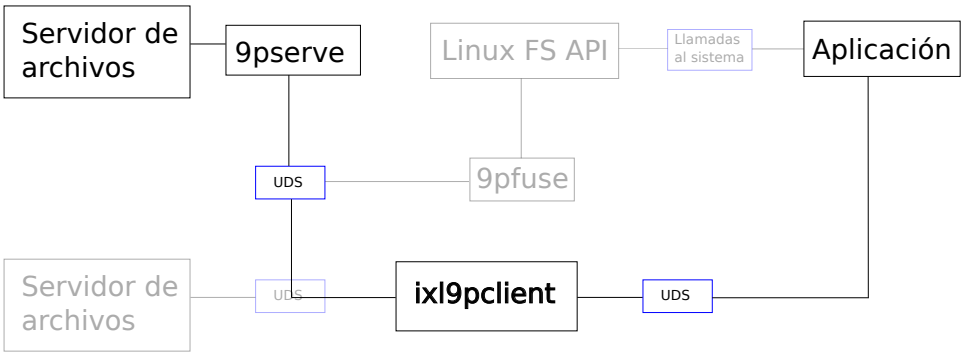
En definitiva, los UDS emplean la interfaz de *sockets* de red[8], pero juega con la ventaja de saber que el proceso al otro extremo **está en el mismo anfitrión**. Así, la implementación no tiene porque utilizar los protocolos de red, sino alguna otra táctica más veloz y sencilla.

Pero además, estos *sockets* responden a las llamadas de lectura y escritura del sistema de ficheros de unix.

Además de servir como sustitutos de las llamadas al núcleo propiamente dichas, eran ideales para realizar la conexión con los servidores de archivos 9P y el servidor central.

El servidor central es, de hecho, el **cliente 9P**.

Pronto se hizo obvio que era posible no romper brutalmente la compatibilidad y tener que reinventar la rueda. Resulta que a «9pserve» le da igual no tener múltiples clientes, porque sigue haciendo de pasarela. En base a eso, este esquema muestra el siguiente paso lógico.



Al lanzar uno de los sistemas de archivos de «plan9port», en lugar de realizar un montaje automático puede establecerse una comunicación en 9P desde cualquier otro programa, lo que es suficiente para interactuar con el árbol de archivos que proporciona.

[8]Que ironía que esta interfaz sea la apropiada para sustentar al sistema de archivos que la vuelve innecesaria.

Uno de los desafíos a la hora de emplear un descriptor de archivos para realizar las llamadas al sistema era que este debía estar disponible sin que el programa se preocupara explícitamente de crear la conexión.

En una de las bibliotecas utilizadas, la función «getixlfd» devuelve una constante donde se ha configurado en ese sistema que se encontrará el descriptor en cuestión. El programa (a través de la biblioteca) juega con ello al bifurcarse.

El problema del establecimiento de la conexión inicial se resolvió mediante un sencillo programa parecido a «exec», que oportunamente sitúa un descriptor abierto en el lugar oportuno.

6.3 El protocolo «sysp»

La biblioteca «libsysp» implementa la cabecera «sysp.h», y su propósito es sustituir las llamadas al sistema clásicas por un protocolo independiente del medio de trasmisión (exactamente como 9P). De esta forma, un mensaje (en forma de *buffer*) puede «volcarse» fácilmente en operaciones de escritura a través de los *sockets*. La mayor parte del tiempo no debe usarse directamente, de hecho solo debería emplearse en dos lugares (en la biblioteca del servidor central, y en el servidor central en sí).

El protocolo y la biblioteca intentan recordar en lo posible a «fcall», especialmente porque hay una sutil relación entre ambos tipos de mensajes, aunque hay algunas diferencias a tener en cuenta. La más importante es que el protocolo no es independiente de la máquina, como norma general solo los campos relativos a los tamaños y los que tienen relación más o menos directa con 9P o están por defecto acotados responden a un número de bytes fijo. Incluso así, es requisito que sea «portable» en implementación. El resto está en manos de la máquina concreta.

El segundo de gran importancia es que mientras que fcall utiliza (y en general, altera) el *buffer* de un mensaje reconstruido para mantener los campos de la estructura, en este caso no hay alteración ni es necesario preservar nada. Sin embargo, una estructura reconstruida emplea memoria asignada (malloc) que debe liberarse mediante la función auxiliar apropiada. Se debe también poner los punteros de la estructura a «nil» antes de la reconstrucción (puede emplearse memset).

Además de que las dos estructuras principales son una unión de estructuras con nombre (a pesar de haber una opción en el compilador para asemejarse al de Plan 9, resulta engorroso y peligroso pretender que en este caso los miembros de todas las subestructuras sean diferentes), resulta que hay una para las solicitudes y otra para las respuestas. Cada una de las primeras debe ser honrada con una contestación, así que es el medio de transmisión y la arquitectura de las conexiones del servidor con respecto a las aplicaciones puede ser «bloqueante».

Pero hay más cosas a tener en cuenta. Las macros empleadas para «serializar» y recuperar las estructuras provienen de la propia «fcall», aunque estas podrían caer en comportamiento indefinido. Peor aún, la falta de signo en 9P con respecto a su presencia en los campos de las llamadas al sistema originales provocan una discrepancia difícil de conciliar. Por ello, la API debería revisarse de nuevo en la segunda versión (incluso si se queda como está).

Por otro lado, la versión original no utiliza el tipo «size_t», algo que va a respetarse[9]. Dado que «plan9port» utiliza algunas funciones de unix, todavía tiene influencia en la implementación y hay que andarse con cuidado.

Para colmo, algunos problemas relacionados con los tamaños y las bibliotecas no pueden solucionarse satisfactoriamente hasta no abordar algunos de los puntos anteriores. No obstante, la alternativa es recrear por separado parte de esta biblioteca. Esto debe evitarse, porque introduciría un tipo de inconsistencia propensa a introducir nuevos errores, y porque la mayoría de problemas se solucionarían en «libsysp» en cuanto se traten en «plan9port».

Un efecto secundario no deseado de las macros con nombres de llamadas al sistema utilizadas por los programas en «plan9port» (pero como ya se explicó, solo mientras no se utiliza NOPLAN9DEFINES) altera algunos nombres de los miembros de cada estructura, aunque realmente no tiene importancia por la forma en la que se usa esta biblioteca.

Los nombres de los miembros no se corresponden «uno a uno» con los de las llamadas, algunos incluso se reutilizan para generar varios mensajes por motivos prácticos. El código que representa situaciones de error solo es válido durante las respuestas.

[9]En el manual de Linux, **accept (2)** termina con un gran párrafo citando a Linus Torvalds bastante divertido, en el que da su opinión abiertamente sobre «size_t».

Para finalizar, hay una particularidad en el protocolo en los mensajes empleados durante el uso de «rfork». Puede parecer por su nombre que esta llamada al sistema pertenece al dominio de los procesos, pero en Plan 9 permite manipular diferentes *recursos* del proceso hijo, o incluso alterar los del original sin crear una copia. Puede verse una explicación detallada y ejemplos prácticos en el libro de Ballesteros, capítulo 7[7].

Algunos de los recursos son el **espacio de nombres**, la **tabla de descriptores de archivo** y los **directorios de trabajo y raíz**, que definitivamente guardan relación con este módulo.

Mientras que considerar que todos procesos comparten estos recursos resultaría trivial, lo cierto es que a veces crean una copia o establecen una nueva instancia en blanco de los mismos, y hay que determinar con que otros procesos los comparten. Pero el problema es que el servidor no sabe nada de los procesos[10]. Para distinguirlos emplea la conexión al *socket*.

Resulta que en aquellos sistemas en los que los hilos son planificados *apropiativamente* hay que tener especial cuidado, porque la conexión al *socket* no debe utilizarse más de una vez al mismo tiempo. En base a esto, tampoco debe compartirse entre procesos.

La solución es dejar que el proceso se bifurque normalmente, pero previamente crear una nueva conexión y validarla como una bifurcación de la actual con las características apropiadas. Para ello, la nueva conexión declara que es una bifurcación, y recibe como respuesta un número *hash*. En un segundo paso, la conexión antigua utiliza ese número para indicar que ha sido bifurcada. Esto hace que el número quede libre, incluso en caso de error.

```
typedef struct Syscall   Syscall;
typedef struct Syscallcreate    Syscallcreate;   /* open, remove */
typedef struct Syscalldup       Syscalldup;
typedef struct Syscallclose     Syscallclose;
typedef struct Syscallpwrite    Syscallpwrite;   /* pread */
typedef struct Syscallseek      Syscallseek;
typedef struct Syscallwstat     Syscallwstat;    /* stat */
typedef struct Syscallfwstat    Syscallfwstat;   /* fstat */
typedef struct Syscallchdir     Syscallchdir;
typedef struct Syscallfd2path   Syscallfd2path;
typedef struct Syscallbind      Syscallbind;     /* unmount */
typedef struct Syscallmount     Syscallmount;
```

[10] Y para complicar todo enormemente, los *sockets* UDS no ofrecen uniformemente (cuando lo ofrecen) un mecanismo de autenticación entre plataformas unix.

```
typedef struct Syscallrfork     Syscallrfork;

typedef struct Sysreply Sysreply;
typedef struct Sysreplyerror    Sysreplyerror;
typedef struct Sysreplyfd       Sysreplyfd;     /* open, create, dup */
typedef struct Sysreplypread    Sysreplypread;  /* pwrite */
typedef struct Sysreplyseek     Sysreplyseek;
typedef struct Sysreplystat     Sysreplystat;   /* wstat, fstat, fwstat */
typedef struct Sysreplyfd2path  Sysreplyfd2path;
typedef struct Sysreplynewrfork Sysreplynewrfork;

/*
        WARNING: set member pointers as nil before to use.
*/
void    freeretC(Syscall s[]);
void    freeretR(Sysreply r[]);

/*
        These functions return zero on error.
*/
ulong   syspsizeC2M(Syscall s[]);
ulong   syspC2M(Syscall s[], uchar buf[], uint nbuf);
ulong   syspM2C(uchar buf[], uint nbuf, Syscall s[]);

ulong   syspsizeR2M(Sysreply r[]);
ulong   syspR2M(Sysreply r[], uchar buf[], uint nbuf);
ulong   syspM2R(uchar buf[], uint nbuf, Sysreply r[]);
```

6.4 La API del sistema de ficheros

Plan 9 tiene una lista de llamadas al sistema sorprendentemente corta. Dado que muchas se pueden además implementar en la biblioteca, la lista de auténticas llamadas al núcleo es incluso menor.

Aunque en este caso no se utilizan llamadas al núcleo, resulta que muchas de las funciones también se pueden hacer a modo de envoltorio. La interfaz del prototipo es la siguiente:

```
int     ixlopen(char file[], int omode);
int     ixlcreate(char file[], int omode, ulong perm);
int     ixldup(int oldfd, int newfd);
```

```
int     ixlclose(int fd);
int     ixlremove(char file[]);

long    ixlread(int fd, void *buf, long n);
long ixlreadn(int d, void *buf, long n);
long    ixlwrite(int fd, void *buf, long n);
long    ixlpread(int fd, void *buf, long n, vlong offset);
long    ixlpwrite(int fd, void *buf, long n, vlong offset);
vlong   ixlseek(int fd, vlong n, int type);

int     ixlstat(char file[], uchar edir[], int nedir);
int     ixlfstat(int fd, uchar edir[], int nedir);
int     ixlwstat(char file[], uchar edir[], int nedir);
int     ixlfwstat(int fd, uchar edir[], int nedir);

int     ixlchdir(char path[]);
int     ixlfd2path(int fd, char buf[], int nbuf);

int     ixlbind(char file[], char mtpt[], int flag);
int     ixlmount(int fd, int afd, char mtpt[], int flag, char *aname)
int     ixlunmount(char *file, char mtpt[]);

int     ixlfork(void);
int     ixlrfork(int flags);

long    ixlnewrfork(int cfd, int flags);
int     ixloldrfork(int cfd, int flags, long hash);
```

Llaman la atención las últimas dos funciones. Estas no provienen de Plan 9, y son una manera de implementar «rfork».

Su propósito es tomar una conexión y convertirla en un derivado de otra existente, y además son públicas para que otras funciones similares a «rfork» puedan obtener fácilmente su propio envoltorio. Para ello, el servidor central asigna un número (y unos flags) a la conexión que invoque a «ixlnewrfork» (preferentemente una conexión nueva), y se lo devuelve.

Si dicha conexión se creó para un «rfork», significa que el proceso al que pertenece todavía no se ha bifurcado, y está esperando a recibir el número e introducirlo en «ixloldrfork» para asociarlo a su conexión original.

Cuando el servidor central recibe esta llamada, busca a la conexión con el mismo número y flags, se deshace el número (ya no espera ser bifurcada) y modifica su espacio de nombres o la tabla de montajes apropiadamente.

Por otro lado, si se desea manipular a la conexión propia, puede hacerse mediante una sola invocación a «ixloldrfork» omitiendo la bandera «RFPROC» y con el *hash* a cero (el servidor no asigna este valor).

6.5 Arquitectura del servidor

Esta clase de programas saca provecho de la biblioteca de hilos (THREAD(3)), y también de las estructuras de tipo *lock* (LOCK(3)).

Como esta clase de hilos son planificados cooperativamente[11] (a diferencia de los procesos de la misma biblioteca, que comparten memoria con el que los crea), muchas de las operaciones que normalmente implican condiciones de carrera son evitadas cuando transcurren dentro de ellos.

Esto puede usarse para liberar las estructuras de datos mediante un contador de referencias representado por un simple número entero. La aparente desventaja es que en caso de bloqueo el proceso durante las operaciones de entrada y salida, los hilos también se detienen.

Pero la biblioteca ya prevé esta circunstancia, y permite crear un proceso auxiliar que realice cualquier tarea capaz de bloquear al programa (IOPROC(3)), y que ejecuta las funciones que el hilo correspondiente le indique (no debe solicitarse más de una a la vez). Al realizar esto, el hilo que ha realizado la llamada libera la CPU en espera de que la función se complete y retorne, permitiendo a los otros hilos sequir con sus responsabilidades.

Además de incorporar ya algunas rutinas de amplio uso versionadas para la ocasión («ioread», «iowrite», «iodial», «iosleep», etcétera), «iocall» permite fabricar nuevas.

Por ejemplo, el servidor central realiza las operaciones complementarias a «dial» en la comunicación con las aplicaciones mediante «announce», «listen» y «accept». Para que se puedan aceptar nuevas conexiones entrantes mientras el resto del programa avanza, «listen» no debe ser bloqueante (y de paso, se puede evitar el uso directo de «yield»).

[11] Esto significa que los propios procesos (o los hilos en este caso) liberan la CPU para que otros puedan usarla, mediante operaciones como «yield». La planificación apropiativa, en cambio, expulsa al hilo o proceso cuando lleva cierto tiempo en ejecución, sin su intervención.

Por otro lado, «QLock» permite *encolar* a los procesos e hilos que quieran acceder a una zona de exclusión mutua en un estado de suspensión hasta que esta queda libre. Al usarse con «Rendez», un hilo puede esperar que se cumpla una determinada condición, y despertará completamente al liberar el *lock* (que adquiere de inmediato). Siempre que dicho *lock* sea el mismo que protege a la condición, no se producirá inanición[7].

Por último, «9pclient» es una biblioteca de gran utilidad, porque facilita la comunicación con los servidores 9P sin tener que hacer uso directo del mismo (9PCLIENT(3)). Esta crea dos hilos para leer y escribir sin bloqueo.

Esto es importante, porque un servidor puede decidir no responder una solicitud de transmisión hasta que se cumpla cierta condición, pero debe seguir siendo capaz de recibir nuevas solicitudes (a diferencia de «sysp»). Para que esto sea posible, y dado que los hilos invocadores se bloquearán, se requiere atender a las peticiones en varios de ellos separados.

La arquitectura se basa en el uso de un hilo de escucha («listenthread») capaz de lanzar nuevos hilos de conexión a clientes (del servidor central) cada vez que uno de ellos intenta conectarse.

Estos hilos se comunican mediante una tabla global de tamaño fijo (configurable antes de la compilación) que permiten identificar a cada cliente por su conexión en lugar de su proceso. Esto se ha realizado así porque el paso de credenciales mediante los *sockets* de unix no funciona igual en cada plataforma (y eso cuando el sistema concreto utilizado dispone de uno), y por tanto no es *portable*. Al terminar la conexión, una entrada es liberada. En caso de llenarse, el hilo lanzador esperará que una conexión termine.

Aprovechando que el protocolo «sysp» si es bloqueante, el hilo de la conexión bloquea la estructura donde se debe situar la respuesta, lanza la petición a las entrañas del servidor (junto con la conexión, que incrementa su contador, todo dentro de una estructura «Call») y duerme en el «Rendez» esperando que esta sea atendida. Así, excepto en caso de *bug*, se sabe que solo puede haber una petición asociada a esa conexión al mismo tiempo en todo el servidor, y eso solo si el cliente espera respuesta.

Otro proceso se encarga de tomar una petición y atenderla (si puede) o enviarla (si fuera necesario). Este *planificador* «inputdispatcher» utiliza una única cola, y aprovecha el funcionamiento de QLock para atender a los hilos de conexión de uno en uno.

```
struct RQueue{
        QLock    lck;
```

```
        Rendez  empty;
        Rendez  full;
        void            *e;
};

void
rqput(RQueue *q, void *e)
{
        assert(e);
        qlock(&q->lck);
        while(q->e)
                rsleep(&q->full);
        q->e = e;
        rwakeup(&q->empty);
        qunlock(&q->lck);
}

int
rqcanput(RQueue *q, void *e)
{
        assert(e);
        qlock(&q->lck);
        if(q->e)
                return -1;
        q->e = e;
        rwakeup(&q->empty);
        qunlock(&q->lck);

        return 0;
}

void *
rqget(RQueue *q)
{
        void            *e;

        qlock(&q->lck);
        while(!q->e)
                rsleep(&q->empty);
        e = q->e;
        q->e = nil;
        rwakeup(&q->full);
        qunlock(&q->lck);
        assert(e);
```

```
        return e;
}
```

Una implementación basada en el encolamiento de mensajes en lugar de peticiones no resultaría ventajosa, porque tanto las semánticas de los «Rendez» (primero en despertar) como el planificador cooperativo favorecen a que el que recoge los mensajes actúe en cuanto el primer mensaje sea encolado.

Como de todas maneras los procesos deben estar inactivos hasta obtener respuesta, es mejor que una vez que se tiene éxito el hilo recupere el control al instante (y se detenga a esperar). En cualquier caso, debe tenerse especial cuidado en que el hilo duerme una vez el mensaje circula por el servidor central.

Cuando el planificador encuentra un error (o descubre una tarea que puede atender directamente) al examinar la llamada al sistema, este genera un mensaje de réplica dentro de la estructura de la conexión y despierta al hilo en suspensión.

```
struct ClientConn{
        int     ref;
        int     fd;
        QLock   lck;
        Rendez  empty;  /* (condition) */
        Sysreply        r;      /* just here */
        ...
};
```

En caso contrario, las estructuras se pasan a un servidor 9P mediante la tabla correspondiente de conexiones y la biblioteca «9pclient», lo que permite que un segundo planificador termine la tarea de escribir la respuesta desde otra cola RQueue.

6.5.1 Enlace con los sistemas de ficheros

Para unir las llamadas al sistema, las conexiones, los espacios de nombres y a «9pclient», se necesita almacenar unas tablas de montajes y de archivos abiertos.

Antes de nada, los archivos son referidos en forma de «canales» («Chan»). Las operaciones del sistema simplemente abren un canal o lo manipulan hasta obtener el resultado esperado. Por simplicidad, se prescinde de los *dispositivos*.

Estos canales contienen la ruta absoluta del archivo que describen (un nombre del canal, o «cname»), y aunque esta no es del todo fiable (cambios en el espacio de nombres) permiten atravesar el sistema de archivos en busca del servidor correspondiente.

Para ello, las conexiones tienen una **tabla de montajes** y un **grupo de descriptores de archivo** («Mtab» y «Fgrp»), que pueden compartir o no.

Las tablas de montaje consisten en un vector de puntos de montaje («Mtpt») concatenados, cada uno con una lista de archivos montados («Mount»). Estas estructuras almacenan un canal (en «Mtpt» denominado «from», y en «Mount» denominado «to»), de tal manera que el primero se convierte en alias del segundo. Debe asegurarse que ningún punto de montaje es, a su vez, montado.

Cuando se quiere alcanzar una ruta, esta primero es simplificada con ayuda de «cleanname» (ya implementada en la biblioteca) y luego analizada elemento a elemento en comparación constante con la tabla de montajes para saber cuando debe resolverse otro archivo.

Por su funcionamiento, la tabla de montajes y la tabla de descriptores no se referencian mutuamente, y es perfectamente factible que una conexión tenga acceso a un archivo fuera de su espacio de nombres que ya estuviese abierto con anterioridad.

Las llamadas al sistema se traducen en operaciones de manipulación sobre el espacio de nombres, y de interacción con los árboles de archivos. Para ello, «9pclient» asocia a cada servidor 9P una estructura «CFsys» que lo representa. Después, las funciones generan **fid** («CFid»), que juntos permiten asociar a un canal con un archivo.

Cuando ya no es necesario un identificador de archivo, este puede liberarse mediante una operación de *clunk*. Los fid no son únicos, y no pueden utilizarse para realizar comparaciones entre dos canales. Para ello se almacena un **Qid**, solicitado al servidor 9P, que es el que los asigna.

6.5.2 Multiplataforma

El código ha sido probado en una máquina FreeBSD 10-2-RELEASE para amd64 con «clang». Es necesario hacer unos ajustes en plan9port (2014-03-06) durante la instalación[10].

El programa INSTALL tiene una referencia a «gcc» de debe sutituirse. El enlazador hace lo mismo, y no enlaza correctamente las versiones posteriores a la novena. En cuanto al compilador, también se debe modificar, ya que el *case* donde se detecta cual está instalado falla y hay que resolverlo desde la opción por defecto.

Capítulo 7

Conclusiones

El módulo de la tabla de montajes no es completamente funcional para el día de la entrega (seguramente estará terminado algo después). Sin embargo, las interfaces están bien probadas mediante programas independientes. El código incluido en el CD permite a las aplicaciones conectarse al programa central, e intercambiar mensajes en «sysp», una pequeña corrección en esta biblioteca, y algunos fragmentos del módulo incompleto.

Este programa permitirá conectar árboles de archivos sintéticos, que provengan de los programas como una interfaz natural. La compatibilidad con los servidores de «plan9port» ofrece todo un campo de pruebas.

Algunas carencias pueden ser subsanadas al incluir otros módulos y aspectos del sistema, tales como la conexión de red o la seguridad. Esto debería ser fácil gracias al bajo acoplamiento con estos, que tiene origen en el enfoque distribuido del diseño original. Ya que no se puede delegar todo en un núcleo local y aún así es necesario intercambiar datos y proporcionar servicios de forma fiable, hay que buscar otra manera de conseguirlo más flexible y general.

Por otro lado, crear programas que aprovechen este diseño (como Acme o Rio) es un reto interesante. Y antes de eso, una colección de programas del sistema complementaría perfectamente a los servidores de «plan9port».

Además de fortalecer este sistema base, es una buena oportunidad para comenzar a desarrollar un verdadero sistema capaz de ejecutarse directamente sobre el metal, e incluso desarrollar las primeras aplicaciones en este entorno.

El repositorio del programa se puede encontrar en Github:

https://github.com/mikel-os/ixl

Bibliografía

[1] R. Pike, D. Presotto, K. Thompson, and H. P. Trickey, *Plan 9 From Bell Labs*. 1990.

[2] N. Stephenson, *En el principio... fue la línea de comandos*. Proyecto Editorial Traficantes de Sueños, 4 2003.

[3] B. Kernighan and R. Pike, *The Unix Programming Environment*. Pearson Education, 2011.

[4] A. S. Tanenbaum and A. S. Woodhull, *Operating Systems Design and Implementation (3rd Edition)*. Pearson, 3 ed., 1 2006.

[5] R. Pike, *Lexical File Names in Plan 9 or Getting Dot-Dot Right*. 2000.

[6] *Plan 9 Programmer's Manual Volume 1*. 2002.

[7] F. J. Ballesteros, "*Introduction to Operating Systems Abstractions Using Plan 9 from Bell Labs*." https://lsub.org/who/nemo/9.intro.pdf, 2007.

[8] "*Mouse vs Keyboard*." plan9.bell-labs.com/wiki/plan9/mouse_vs._keyboard/index.html.

[9] "*FUSE API documentation*." https://lastlog.de/misc/fuse-doc/doc/html/.

[10] M. van Atten, "*Building on FreeBSD 10.1 after commit ...*." ttps://groups.google.com/forum/#!msg/plan9port-dev/bjd3SFMuT9U/X4PWu42UUWIJ.

[11] F. J. Ballesteros, "*Notes on the Plan 9™ 3rd edition Kernel Source*." https://lsub.org/who/nemo/9.pdf, 2007.

[12] *install(1) plan9port*. https://swtch.com/plan9port/man/man1/install.html.

[13] "*Notes on Programming in C.*" http://doc.cat-v.org/bell_labs/pikestyle.

[14] *style(6) Plan 9.* http://man.cat-v.org/plan_9/6/style.

[15] *9pclient(3) plan9port.* https://swtch.com/plan9port/man/man3/9pclient.html.

[16] *bio(3) plan9port.* https://swtch.com/plan9port/man/man3/bio.html.

[17] *dial(3) plan9port.* https://swtch.com/plan9port/man/man3/dial.html.

[18] *fcall(3) plan9port.* https://swtch.com/plan9port/man/man3/fcall.html.

[19] *ioproc(3) plan9port.* https://swtch.com/plan9port/man/man3/ioproc.html.

[20] *lock(3) plan9port.* https://swtch.com/plan9port/man/man3/lock.html.

[21] *notify(3) plan9port.* https://swtch.com/plan9port/man/man3/notify.html.

[22] *thread(3) plan9port.* https://swtch.com/plan9port/man/man3/thread.html.

[23] *0intro(4) plan9port - introduction to file servers.* https://swtch.com/plan9port/man/man4/intro.html.

[24] *9pserve(4) plan9port.* https://swtch.com/plan9port/man/man4/9pserve.html.

[25] *Sección 9p del manual de plan9port.* https://swtch.com/plan9port/man/man9/intro.html.

[26] *unix(7) Linux.* http://man7.org/linux/man-pages/man7/unix.7.html.

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR