A dream of an ultimate OS

Oleg Kiselyov
CIS, Inc & University of North Texas
303, N.Carroll, Suite 108 Denton TX 76201
oleg@ponder.csci.unt.edu, oleg@unt.edu, http://replicant.csci.unt.edu/~oleg/ftp/

This is a dream, and as such, it is made of shreds of reality shuffled and rearranged in sometimes bizarre combinations. It has been brewing on for over 10 years out of dissatisfaction with many of the major modern Operating Systems. Indeed, it is glaringly obvious that the only thing a user does at the terminal is requesting/reading/modifying textual information, arranged mainly into tables/scrollable lists. Nevertheless, the user often has to use different and completely disparate commands to achieve exactly the same modification, for example, to delete an item (line) from a list as in: removing a line in a text document, removing a file, killing a process (that is, removing it from the list of active processes) and canceling a print job. Furthermore, despite the fact that the OS is swarmed with tables – from a hierarchical database of files to yellow pages maps, to a hash dictionary of an object file archive (library), to relatively flat databases of IP routes, current processes, users, and code revisions – common database functions like inserting a record into a "table" with hashing a key, retrieving a record/field using a simple/concatenated key and linking tables, are conspicuously missing among the core kernel services. This paper is an attempt to imagine what an OS would look like and how it would work if looking for a word 'foo' in something and deleting/closing/stopping this something, be it a paragraph of text, a network connection, a subscribed newsgroup, a process – would all require roughly the same sequence of mouse clicks/keystrokes, and would be understood and interpreted in the same spirit by the operating system.

Introduction

Operating Systems are commonly viewed from two major standpoints: managing computing resources, and hiding hardware idiosyncrasies while putting a *friendly* face for a user. IMHO, providing a user-friendly interface is the **primary** OS responsibility: after all, a CPU doesn't need any OS to run, nor does it care much which code, system or user, it currently grapples with. Nevertheless, with a notable exception of MacOS, MagicCap and Newton's OS, operating systems have been thrusting upon a user a flood of disparate interfaces, commands, and actions – all are for doing essentially the same thing: filling in and editing some list/table. Moreover, even internally every major OS component – a file system, network services, user management, terminal management, to name a few – each implements and manages its own private incarnation of a simple database.

It literally springs to mind that database services and text/list editing are a core activity ought to be supported on the very fundamental level of OS. The paper flashes a few images reflecting some particular examples of how that unification could be done and how one can work with it. Here is the preview:

MacOS is one of the very few systems around close to the ideal: TextEdit has been elevated to the level of a standard system (toolbox) service. It was the strong statement that the OS is not only about managing files and processes. Furthermore, deleting a piece of text, a file, a directory, a file server connection – all can be accomplished by the same action: highlighting and dragging into trash. Still, one can go a little bit further: for example, a list of processes conceptually isn't much different from the list of files. One can imagine the Finder manage (arrange, get info, duplicate, trash) files and folders that are not necessarily ordinary files and folders, like processes, open TCP connections, newsgroups, print jobs active and pending, to-do-tasks, etc.

Deep down, an operating system is nothing but a manager of many databases. Indeed, a file system, the process table, routing tables, list of known AppleShare servers, revision control system (projector) data, Think C projects - they're all databases. Unfortunately, despite a sizable share of common functionality and interface, every one of them is implemented and managed separately. For example, hashing, searching, cashing are used in many of those databases (if not all of them), but each application implements and re-implements them in its own way. One would think that if a database manager were a universal **core** program, one could take trouble really polish hashing and caching, once and for all. It really looks like a very good deal to trade a multitude of "custom" database managers for one well-designed distributed database manager.

Conventional databases are usually implemented on the top of a file system. But the file system itself is a database. Mac's HFS and Novell's file system even use btrees and other "advanced" indexing schemes typically found in full-blown "industrial strength" databases. For another thing, a hierarchical model is not the only db model out there. So, why not to have an Adabas-like or DB4-like or Oracle-like database **instead** of the file system? After all, querying the database for a Jan 1994 sales, and clicking on folders "Sales", "1994", "January" are very related activities. Only, a full-blown database provides a more flexible interface for linking and querying records. Modern databases have all facilities for storing images, sound files, movies, etc. big binary objects. I wonder, what else do we need files for?

Unification of user interface and the underlying database functionality has one more important advantage: ease of linking or mixing together, objects. I imagine documents made of not only chunks of text, but folders and applications themselves (or links to them). Indeed, a document with a picture usually has the text and the image in separate places, with additional information (link) as to how and where the image should be displayed. Why not to store likewise links to menus, applications, or remote servers, or precompiled headers, or *mailto:* or other *form* or *anchor* as well? The desktop is probably going to look just like a ***homepage***.

The rest of the paper shows how one can come to these points and where he can go from there.

Everything is just editing

It is indeed. A document I'm editing is a list of lines; if one edits a file in emacs, he presses \C-K (I press PF4) to delete a line. When we list a contents of a directory, we also get some sort of table or list (especially using ls -l or view-by-name). To remove a line from this table, we use a different command, rm. When we want to see which processes are currently running, we use ps, or top, or ProcessWatcher. Again, we end up with a table, but now we need to use kill process_id (not even a process name) to "delete a line" from the table. To delete a line from the table of queued print jobs, we use yet another command, lprm print_job_id.

Thus deleting a line, removing a file, or killing a process or a route or an ARP entry all boil down to essentially the same thing: removing a row from some table. This is no coincidence. The uniformity runs very deep, both on the level of implementation, and on the level of presentation. Indeed, there is not so many ways how one can manage a collection of objects: it is usually some kind of list or tree. Also, there is not so many different tricks how this collection can be presented to the user and be manipulated by him. The only way people work at a terminal is browsing and editing, that is, moving the mouse, typing and pressing the PgDn key, if you think of it. Disparate interfaces are not a consequence then of some fundamental differences in user or system activities, they are simply result of evolution: different subsystem/services were written by different people and modified by even bigger crowd.

Macintosh definitely stands out of the crowd. Many functions within MacOS are accomplished by exactly the same action (say, removing is by dragging into trash, opening is by double-clicking). This is especially true with a Drag&Drop Manger installed. Even UNIX is moving towards some unification, take for example CDE, or a proc filesystem. The latter is really neat: indeed, with UNIX approach "everything is a file" one could only wonder why a process should be any different (and why it took so long to implement and popularize this idea). However, the unification isn't complete. While it is possible to open /proc/1024 to get hold of a process with id 1024 (just to find out who owns this process and when it was created, if for nothing else), one cannot rm /proc/1024 to kill the process, and one cannot do ls /proc/1024/open_files to see the list of all open files for this process. Though why not?

Since the list of processes conceptually is not much different from the list of files, why don't we have a folder "processes" on a Mac, which has "files" standing for processes. So I could use the standard Finder operations, View-by, GetInfo, Trash, Duplicate to manipulate the processes. Usenet News hierarchy is very similar to that of a filesystem (as a matter of fact, this is how it is stored and managed on an NNTP server). Nuntius newsreader presents the news hierarchy as some directory tree, as folder's and files in a view-by-name mode. Alas, Nuntius had to emulate much of the Finder's functionality to manage these newsgroups-folders. It would be much easier to develop and use, say, printer and network managers, an FTP utility or a newsreader, if one can tell the Finder: here, this is a list of *files*, manage it as you usually do with a list of files, just tell me when you're about to trash something.

Rearranging file icons within a folder view and rearranging paragraphs within the document are essentially the same activity. If it is unified, the overhead and code duplication can be significantly cut down. There is also a good side effect from that: a document can contain folders and/or applications (icons of other applications, etc.). So, we kind of get the hypertext for free.

The luster and dull of plain text

Configuration of a UNIX system is specified and controlled by a huge tangle of *plain text files*, /etc/hosts, sendmail.cf, syslog.conf, inetd.conf, /etc/uucp/Systems to name just very few. .INI files on some other systems are also plain ASCII. Even MacOS caved in a little bit with System Folder:Hosts, although it is a very isolated example on a Mac. Of course, just because symbols displayed on screen must be ASCII, it doesn't mean that the information on disk should be the same. Still, ASCII configuration files abound, for a very simple reason: they can be modified with any text editor from ex and edlin upwards, and can be viewed and created even without an editor, with a cat command. Note, accessing a block of /etc/hosts file is not much easier than reading a /machines tree of Next's netinfo database. ASCII wins simply because tools for handling text files are present even on a naked system. As long as a file system is treated differently than a database, the makers of netinfo, NIS or any other database have to support tools for converting a database table or set of records into a plain text file, and back.

But it does not have to be this way. If a database manager is implemented as a core system service, and the system always comes with a very primitive database access tool, the gordian knot of system configuration files disappears. The situation is almost ideal on a Mac, where ResEdit is this universal database editor. Much (if not all) system configuration can be set up by opening a resource and toggling a few buttons or retyping strings or adjusting colors, without any need to learn syntax of a specific configuration file (and without any need for a system to waste time trying to parse that text file and report an error if any). Unfortunately, ResEdit and a set of templates for system resources do not come bundled with MacOS. But SimpleText always does. That's why System Folder:Hosts is a plain text, though it would have been much better as a resource.

The very idea of an application as a merely collection of code and configuration resources with a common name is beautiful. It is even possible in some applications (LaserWriter Utility, for example) to add/delete menu items and corresponding functionality just by adding/removing appropriate resources, without any need to recompile/relink the application. It was with pain that I read a recommendation to refrain from creating code resources with PowerPC native code (which should be moved to the data fork instead). Now an application has a database managed by the resource manager, and a database managed by a fragment manager.

Everything is Database

And this is not exaggeration: an operating system is permeated with sets, lists, tables, and other collections of something. This something starts with fundamental data structures like a process control block or a page table and continues to include i/o request queues, list of open windows, resources. Also, SCCS, RCS, termcap/printcap, networking databases (name domain service, name resolver, /etc/hosts, /etc/networks, /etc/services, routing tables), whatis database for man pages: one can hardly spend a millisecond working in a computing system without hitting some kind of database query. Note, every of these databases supports basic select/insert/delete functions; advanced options like concatenated keys and linking among several tables are often required, too. Still, almost each database has its own implementation of these common functions. One can argue that managing process control blocks and virtual pages *does* require tailored and very tuned in implementations. However, even a couple of seconds delay in removing a SCCS revision or a host entry is not a big deal at all. Furthermore, many of system databases are implemented as “flat file” with linear search, oftentimes just because it was simpler. Using a well-designed universal database instead of a multitude of quick fixes actually promises a boost in performance (besides, it is just cool).

Having a universal database frees the system or an application from many chores, like time stamping or permission checking, etc. But universality offers another clear advantage: an ability to link all kinds of records/tables, which is so much pain to do now. For example, a link between two records representing files is no different for a universal database manager than a link between a record in the "Users" table, a record in the "Processes" table, a couple of records in the "Files" table and the "Print jobs" table. There is no longer need for multiple IDs and messing with them. Many-to-many links are also possible. Moreover, this improves performance: list of all processes belonging to user joe can be found faster by a database query rather than with ps aux | grep joe. Surely any database can do better than a dumb search, like in netstat -a | grep finger (which is used to finger fingerer). Many similar scripts are just database queries, and not very efficient ones. Makefile would be also easier to generate and maintain. It would also be possible to link #include directly to the includee (which would obsolete the intricate art of specifying compiler’s -I and -L flags and trying to predict which of several possible time.h files the compiler would actually pick up).

File system is a database, and can be a better database

Hierarchical organization of a file system is not the only possible organization. Besides, the widespread use of symbolic links and aliases makes a file system look more like a network database, though kludged (everyone who spent some time reanchoring aliases can tell that). So why not to start with a networking database with good indexing capabilities, something of, say, Adabas? It was very efficient, implementing its own storage management and transparent multi-level indexing; its journaling capabilities are much better than those of UFS (or HFS for that matter). Or take a FileMaker: there are several applications on info-mac for cataloging the contents of CDs and floppies in a FileMaker database. So why a file system could not be like this database from the outset?

Nesting folders is what gives the file system its hierarchical flavor. Directories are just view of a certain subset of files, selected according to some criteria (and named). Thus, a directory can be thought of as a view of database records, a saved database query. It follows immediately that a file can appear in as many “directories” (views) as one wishes to. For example, one can be a view of all files tagged as “sales reports”, another may be a view of files modified within five days. In a sense, “searching” a file system and creating and populating a folder becomes the same activity. Moreover, any file modification would be visible in all views. Since saved views are database objects, one can reference views within views if one so wishes. There is no required hierarchy: you can have two views be referenced from within each other, or create any other network of views that best suits the problem.

Every database record (item) should have some mandatory attributes like time stamps, owner, permissions, kind (document). Beyond that, the user (or application/creator) can add anything he wants. For some records, the body is just a binary object. Records of kind "image" (or in table "images") can have additional attributes like the image width, height and depth, a key of a compressor method, etc. Thus, listing all images 512-pixel wide and with depth at least 8 with a private colormap should be as easy and elementary as viewing files in a folder by date.

Viewing and editing *files* should not be a problem as long as (even a naked) OS has some rudimentary database browser. It may look like a basic no-frills record browser in Paradox (displaying all record fields in columns or as *name=value* pairs). Of course, an OS should have an ability to generate better looking views/reports (as Paradox does). Still, the basic browser is necessary and useful (like cat or SimpleText) in a desperate situation.

Having a general purpose database instead of the file system does not mean that users should forget everything they knew about how to grapple with a computer. Almost all old acquired skills can be used without any change. Indeed, a path to a file in a hierarchical file system is just a list of keys how to locate a file. This becomes especially obvious if one looks at URL, say, http:/somehost/foo/bar.html. This URL does not actually imply that we want to access a file bar.html in the directory foo under the DocumentRoot on the host somehost. If foo is a script, then bar.html is just a parameter passed to that script, and can be interpreted as a file name, or anything else the script wishes. In short, not everything between slashes in URL is a name of a directory. It is just **a key** specifying an object we want to take a look at. The same argument holds for the new file system: one still can locate a file by listing some of its attributes separated with slashes. The advantage of the database approach is that one could put a wildcard in *any* of the “directory” names, or one can consider such file attributes as modification date, size etc. as some “directory names” and use them in the “path”. In a sense, running find and listing a “directory” would be exactly the same activity.

Many industrial strength database managers (Oracle for one) support transparent access to remote databases, with possible local caching to improve performance. It looks then that one doesn’t need NFS at all: distributed database manager takes care of remote mounting, authorization, etc.

Mock-up ‘hello world’ session

It is probably not going to look much different from how we work on a Mac now. Once Mac is powered up, we see a Desktop: a some particular view of the database. It contains references to other database objects like the table of processes, the table of system data (which have attributes System and UserConfigurable: Control Panels in short), the table of recently used objects, etc. Suppose I located a C compiler (well, I wish there were a better language by then, say, Dylan), either on the Desktop, or in one of the subviews, or just doing a plain search for anything with attributes application and C. Once the “C Project Manager” is launched, it prompts as usual to open or create a project. I create a new project and a new file, which is automatically tagged with the project name, C, text, etc. attributes. In a sense, a ThinkC project plays the same role as a directory does now. Of course I can define and set other attributes I wish (some of them could be inferred from the context or from the some other db object). This is like entering data in a database table: one can say, make a new record like a previous one, and then change just a few values. I can also attach a comment to a new C program, and make it as big as I wish. Words from comments can be used in a file search, too.

When writing a “hello world” code itself, I can say "enter a db object", get a query window and find an object with attributes "data, C declaration, owned by system, having something like ‘standard io’ in comments". Or I can say "C declaration for printf()", and hopefully the editor can fill in some default attributes for the query. Of course I can go ahead writing a C code and then click on printf() and tell the editor to find and include some C .h file that declares this function. If the editor inserts something like #include <standard io> I can click on it and see what this package has.

It looks like one can have the functionality like that even tomorrow: doing away with files as we know them does not mean breaking everything, every habit, every skill and every application. On the contrary, most of the old functionality would apply, and some applications even become simpler (the C preprocessor does not need to go into trouble trying to figure out where that .h file is).