A Visual Expedition Inside the Linux File Systems

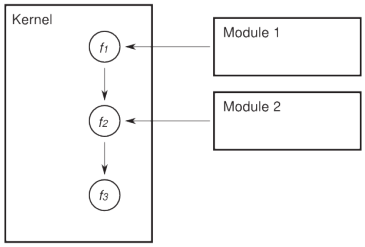
Răzvan Musăloiu-E.

1. Introduction

I photograph to see what the world looks like in photographs.
—Garry Winogrand (1928–1984)

Despite being a very important part of any operating system, file systems tend to get little attention. Linux has three editions for Linux Device Drivers, another three for Understanding the Linux Kernel and two for Linux Kernel Development. For the 2.4 networking stack there is Linux Networking Architecture by Klaus Wehrle et al. and for the memory subsystem there is Understanding the Linux Virtual Memory Manager by Mel Gorman. The aptly named UNIX Filesystems: Evolution, Design, and Implementation is only giving a general overview of how things work. Practical File System Design with the Be File System by Dominic Giampaolo is an an enjoyable read but, as the name indicates, it only deals with BeFS. The same is also true for HFS+ in the very thick but also very interested Mac OS X Internals: A Systems Approach by Amit Singh. I really hope that someday somebody will spend some time and put together a nice book or website in which file systems, new and old, are presented and analyzed in detail.

As the disclaimer from the front page says, I don't know as much as I want about file systems. I'm making progress in learning about them in the traditional way of playing and understanding the existing code. What I'm attempting in this project is to complement this by a visual aproach in which the main purpose is to try to graphically depict some the ways the things go. The main observable thing I'm using is the external symbols used by kernel modules. There are two main reasons for doing this. First, many operating systems build their file systems as kernel modules. This is useful because it separates the part we are interested in from the rest of kernel. And second, because the modules need to be loaded dynamically, the functions they call and data they access from the kernel shows up as external (unresolved) symbols in their binaries. These can be easily extracted using nm. One drawback of this approach is represented by the chains of calls like the ones in the figure from below. Luckily, in the Linux kernel many functions are explicitly marked as inline so this case might not occur so frequently. I haven't check explicitly for this yet to but it is something I would like to do.



Chain of calls problem. Module 1 and Module 2 are two kernel modules and f1 and f2 are two functions exported by the kernel. Even f1 are two f2 different calls they are in fact closely related to each other. This is not captured by the relations between the modules and the kernel and can only be detected by also lookin the way things happen inside the kernel.

That being said, here is a quick overview of the other sections. The following two are the big ones. The first is a detail analysis of one particular Linux Kernel tree and the second is a shorter one done over a big number of file systems from all the 2.6.x Linux Kernels. After that there is a small section that shows some aspects of the BSD family. After conclusions there is an appendix consisting of three things: the first one explains how the file systems for Linux were compiled, the second one shows timelines for the releases of Linux Kernel, FreeBSD, NetBSD and OpenBSD; last is a big detailed map of the external symbols of the kernel modules analyzed in the second section.

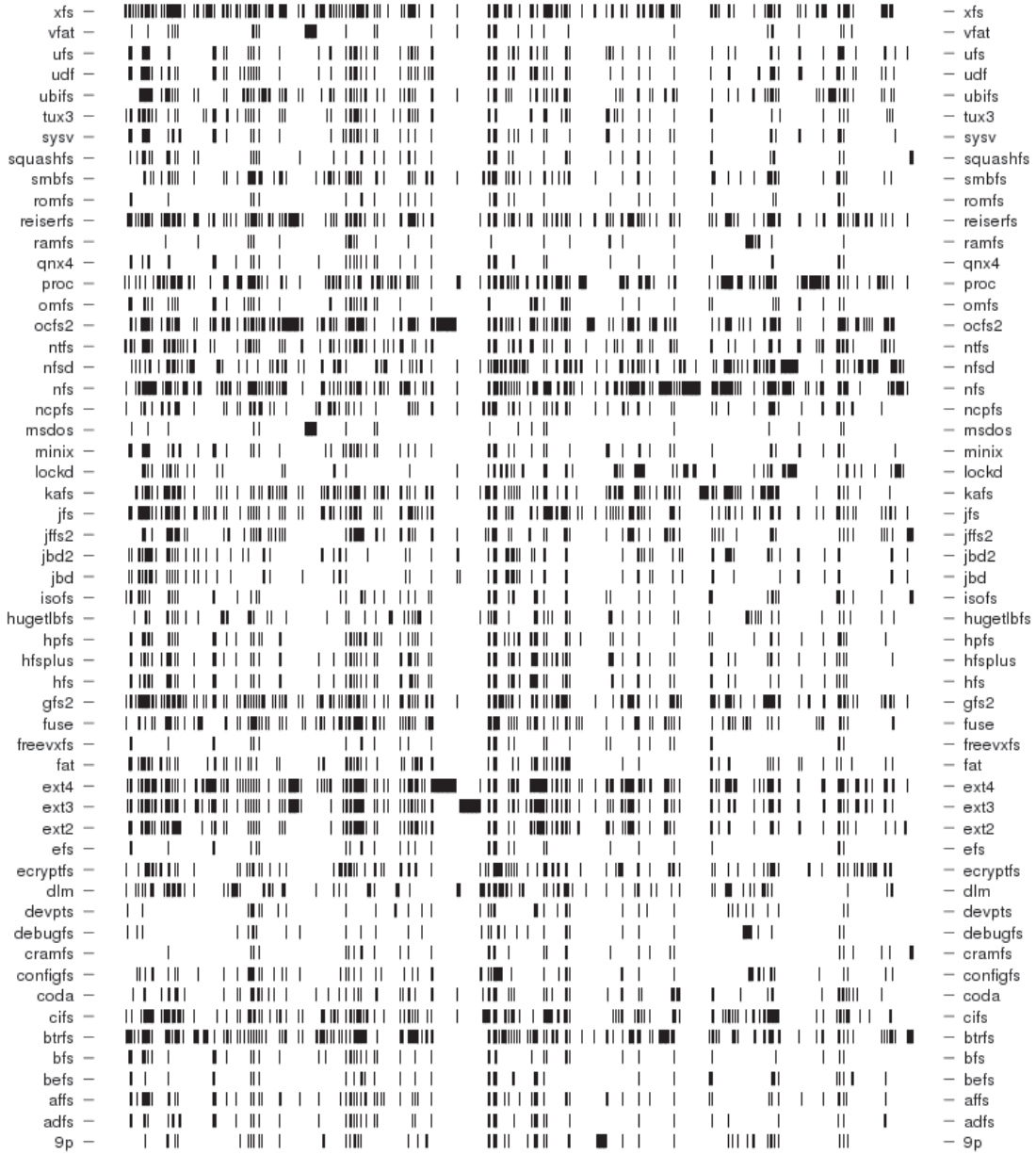
On more thing. The figures are accompany by captions which describe the plot and note some of the interesting things that are going on there. If you find any mistake please let me know and I'll try to fix it.

Happy reading/viewing!

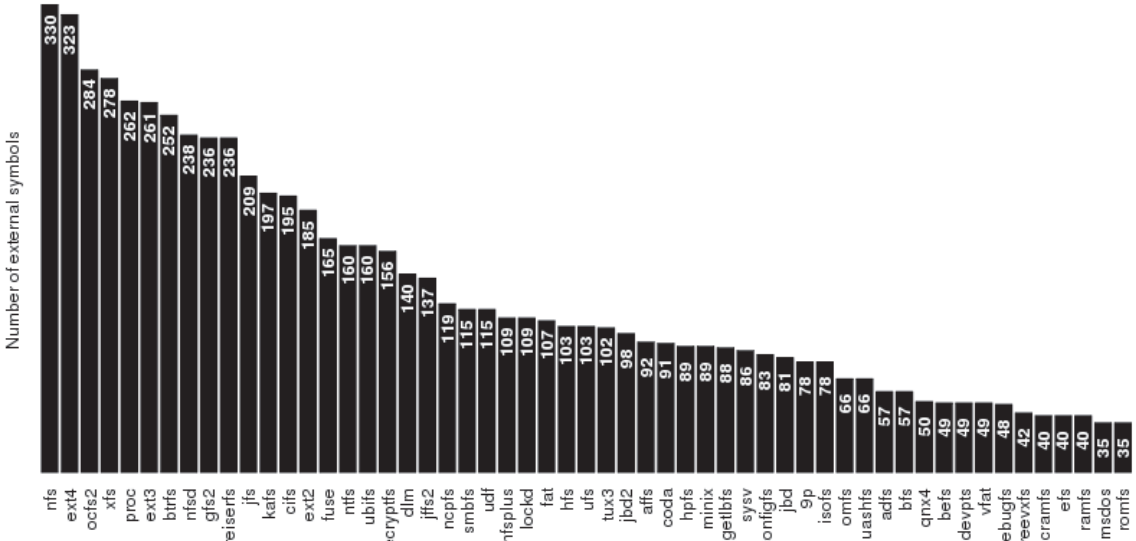
2. Linux Kernel 2.6.29 + tux3

In this section we are going to explore a certain version of the Linux Kernel, the tux3 branch from March 14, 2009. This version contains the Linux tree up to March 10. The 2.6.29 was released on March 23 so this is not exactly the final version. I picked it because it contains btrfs and, at the time of writing this report, it is the latest one that was published by Daniel Phillips, the creator of the tux3.

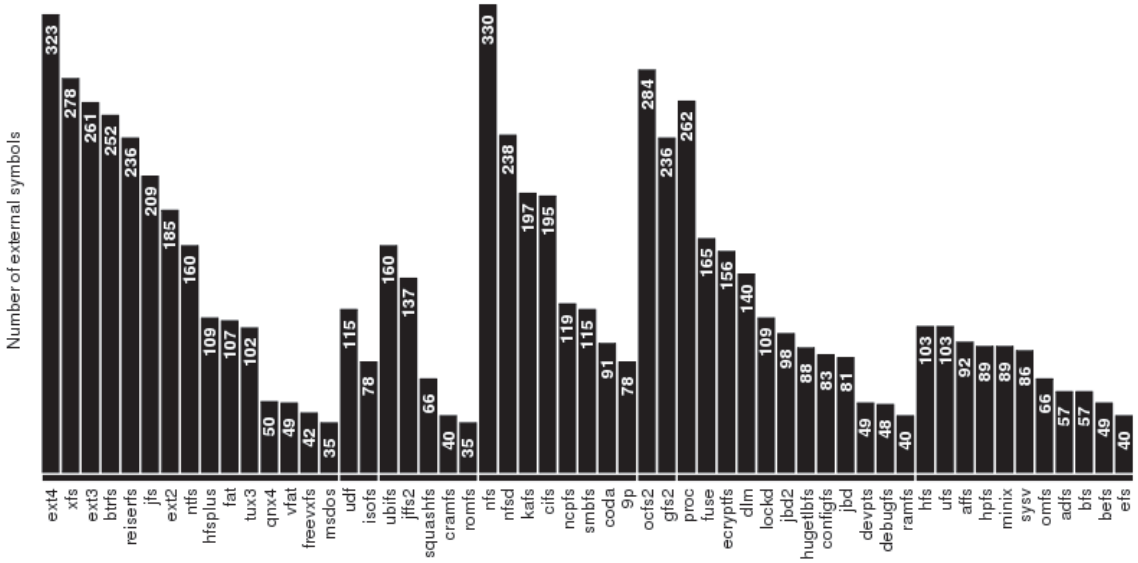
The rest of this section is made up exclusively of figures accompanied by extensive captions.



Map of the external symbols. This plot shows the external symbols for 55 kernel modules from the tux3 git repository. Each tick represents an external symbol. The file systems are show in alphabetical order. The big compact chunk of external symbols related to jbd and jbd2 are visible for ext3 and ext4. The fact that ocfs2 is also making use of jbd2 is also noticeable. Later in this section we will see how this representation looks when is reorganized based on similarities between file systems.



Number of external symbols. This plot shows the ranking of the file systems based on the number of external symbols. We can see that the range is about 300 symbols and there are no sudden jumps: some sizes are more popular than others but overall the space is filled quite smoothly.



Number of external symbols by categories. This is the same plot as the previous one except this time the file systems are grouped in categories.

The first group, which contains the disk-based file systems, is leaded by ext4 which is ahead of xfs by 45 external symbols. Note that the number of external symbols for both ext4 and ext3 is boosted by the fact the journaling part not implemented internally but provided by jbd2 and jbd respectively. At the other end of the scale is a group of 4 file systems out of which only two, freevxfs and qnx4 are truly self-contained. The other two, msdos and vfat are getting most of their functionality from the fat module which is more than twice of their size.

The second group contains the file systems dedicated to optical mediums and it holds no surprises: udf is ahead of isofs by a comfortable margin.

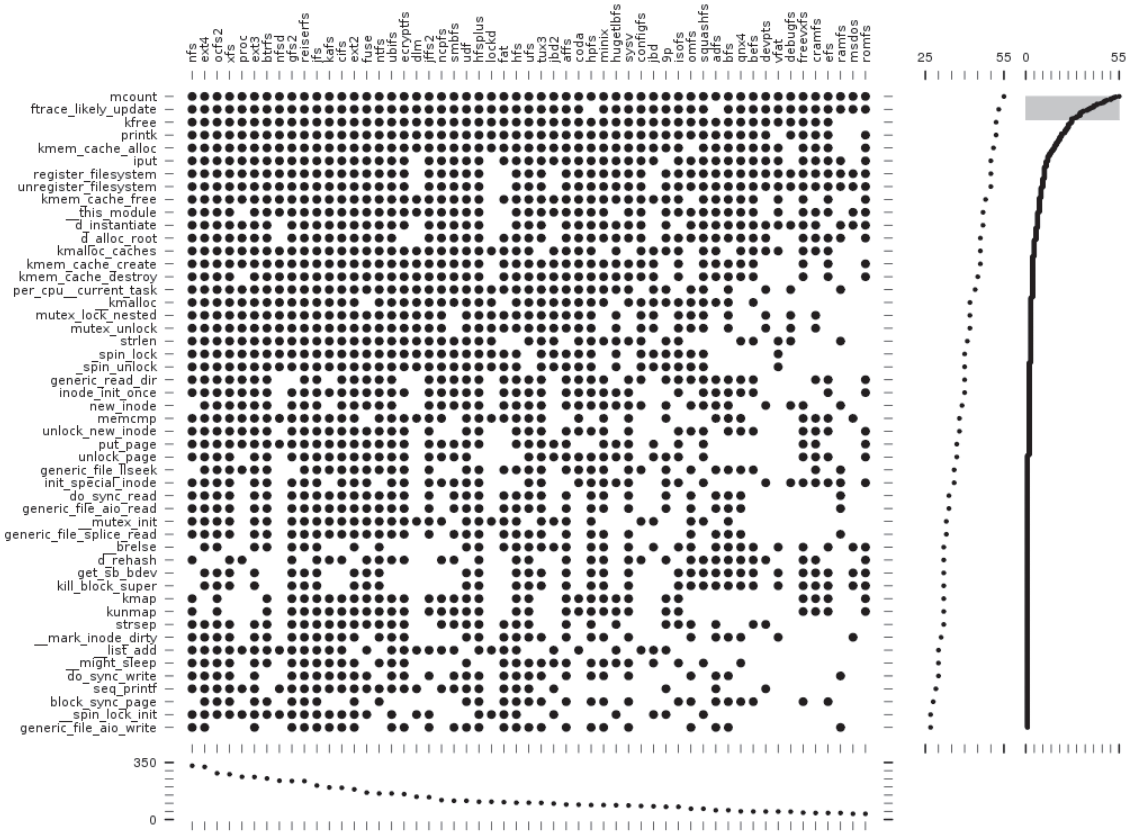
The same thing is also true for the the third group, of the flash-based file systems, where the first place is taken by ubifs followed by jffs2. The third placed is secured by squashfs while the bottom is shared by cramfs and romfs which are separated by only 5 symbols.

The fourth group is the one dedicated to the network file systems. Here the first two spots are taken by nfs and nfsd, which provide kernel support for NFS client and NFS server. On the next two places, at very close distance between them, are kafs, the Andrew File System, and cifs. The end is shared by coda and 9p.

The fifth group contains, in this order, the only two cluster-based file systems: ocfs2 and gfs2. The number of external symbols for ocfs2 is increased due to the use of the jbd2 journaling library.

The sixth group, the one dedicated to memory-based file systems is dominated authoritatively by proc which has almost 100 more external symbols than fuse, the file system from the second places. As expected, at the bottom sits ramfs.

The seventh and last group is dedicated to ancient file systems. The first place is shared by hfs and ufs. Quite surprising, this is only one of the three ties in this group, the other two being hpfs/minix and adfs/bfs.



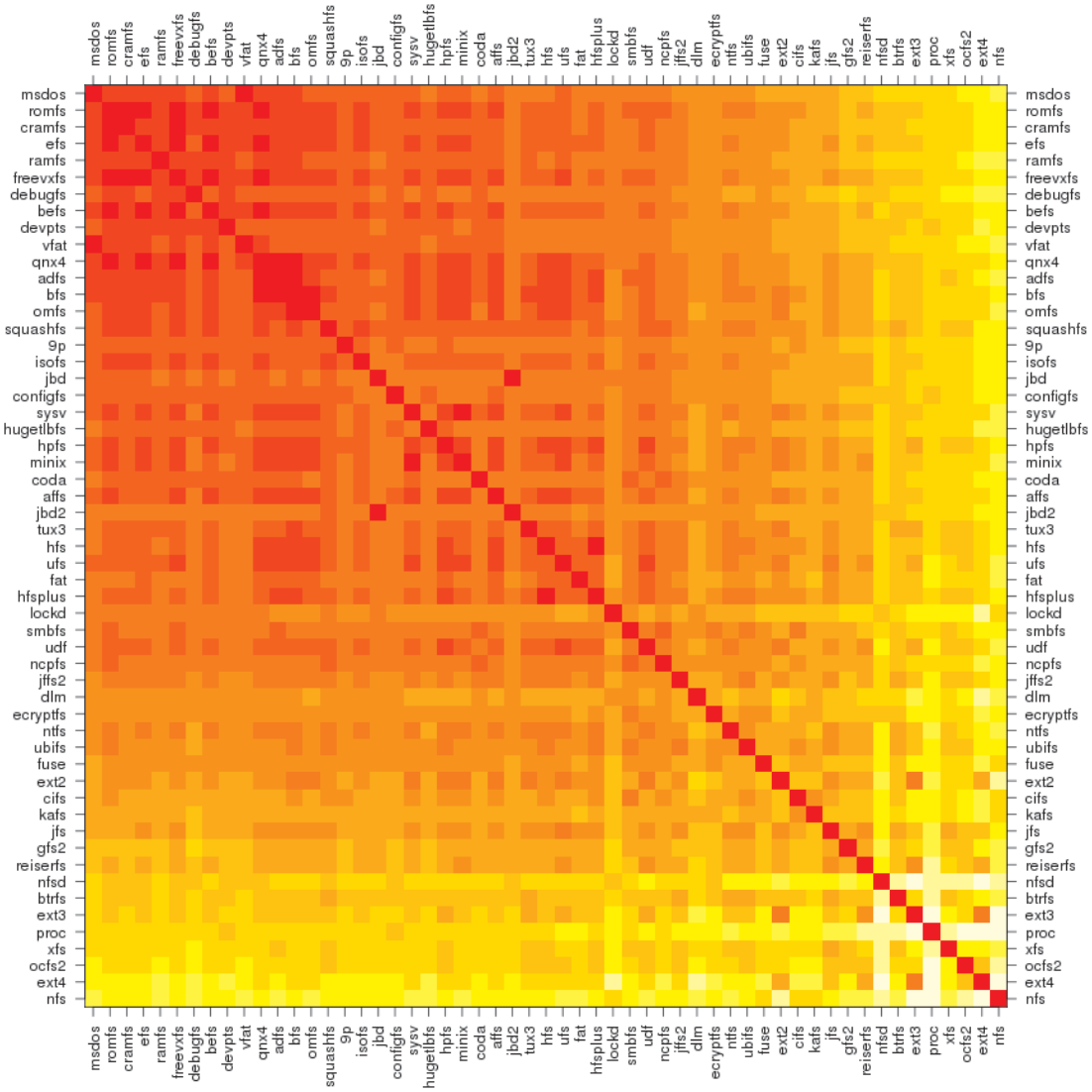
The 50 most popular external symbols. The symbols are sorted in the descending order of their frequency while the file systems are sorted in the descending order of number of external symbols they use.

The first two symbols are used by the function tracer. On the third place we have a tie between kfree, which is used by everybody except by ramfs, msdos and romfs, and kprintf which, surprising, is avoided by vfat, ramfs and msdos.

As expected, the kmem operations are among the most popular one. So are the some basic operations like strlen, memcmp and strsep.

Another thing we can notice are some (expected) pairs of calls that are always use together: register_filesystem/unregister_filesystem, _spin_lock/_spin_unlock and kmap/kunmap.

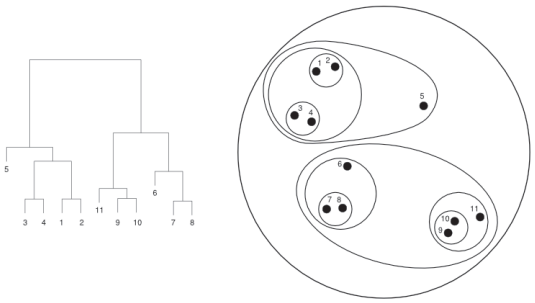
Another (again expected) observation is that the lack of (un)register_filesystem identifiers in the modules which only provides services to others: dlm, lockd, fat, and jbd2/jbd2.



Heatmap of the Hamming distances. If we think of each file systems as a string of bits with one indicating the presence of a certain external symbol and zero as the absence of it then the Hamming distance is the minimum number of substitutions necessary to change one string into another. The heatmap is symmetric. The red indicates a Hamming distance of zero, which corresponds to maximum similarity, and yellow indicates very little similarity. Everyone is similar with itself so the diagonal is red. In this plot the file systems are sorted in descending order of their number of external symbols. What we are going to do next is to reorder the rows and columns in a way that brings closer the file systems that are similar.

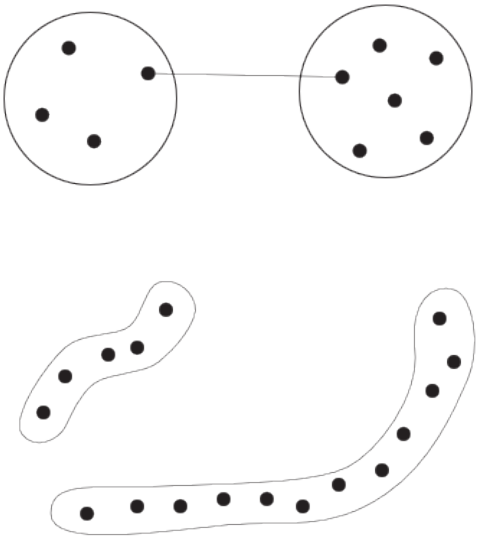
Hierarchical clustering. As the name implies, hierarchical clustering builds a hierarchy of clusters. There are two ways to do this: one is to to start from bottom, with all the data points as clusters and then, at each step, merge two of them. This is also know as agglomerative nesting. The other one is called divisive and starts from top, with everything in a big cluster, and at each steps performs a split.

At the right is an example of clustering 11 points situated in a 2D plane. Left is a dendrogram, a tree diagram usually used to represent the result of a hierarchical clustering. Right is a a representation using nested clusters.

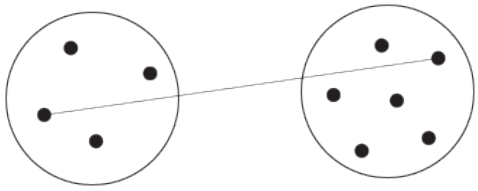


Single linkage. When all the clusters only contains one point we can easily define the distance between them as the distance (d) between the respective points. After each merge operation we need a way to define the distance (D) between this new cluster and all the old ones. One way to do this is to consider the distance between two clusters as the minimum distance between any pair of nodes with one node in a cluster and another one in the other. Formally we could express this as D(A,B) = { min(d(x,y)) ∣ x in A and y in B }.

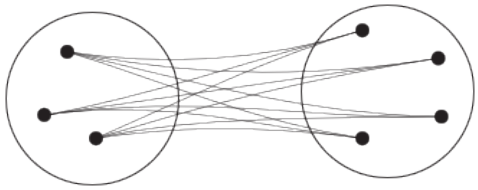
This method is best suited for constructing elongated clusters like the ones int he right.



Complete linkage. In this case the distance between clusters is defined as the maximum distance between the pairs of nodes which contains one node from one cluster and one from the other. Similar with the previous case, this could be express as D(A,B) = { max(d(x,y)) | x in A and y in B }. This method is capable of creating small and compact clusters.

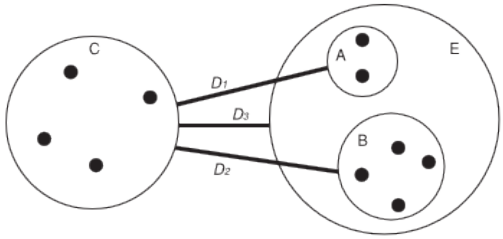


Group average. We can also define the distance between two clusters in such a way that all the pairwise distances contribute to the result. If we take the average of all the pairs then the method is called group average.

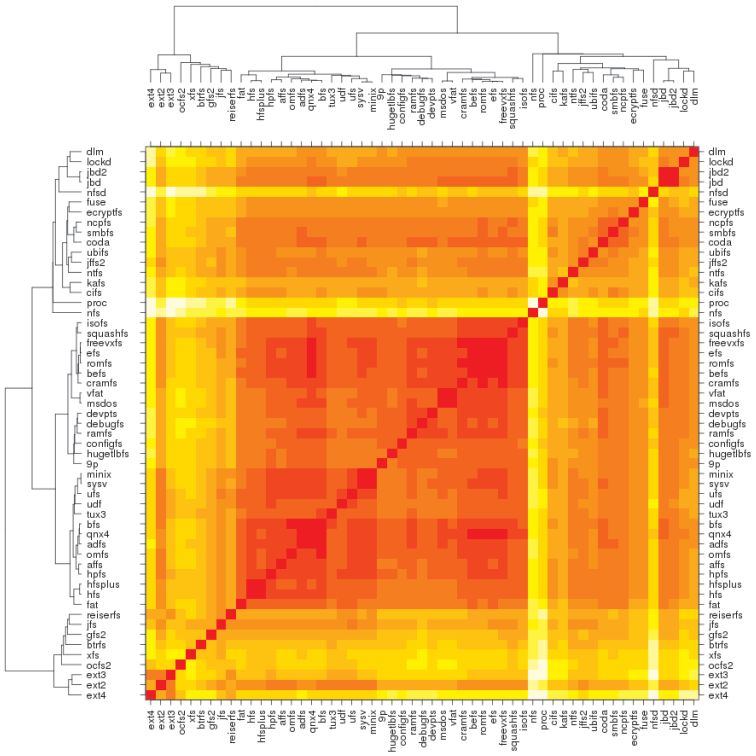


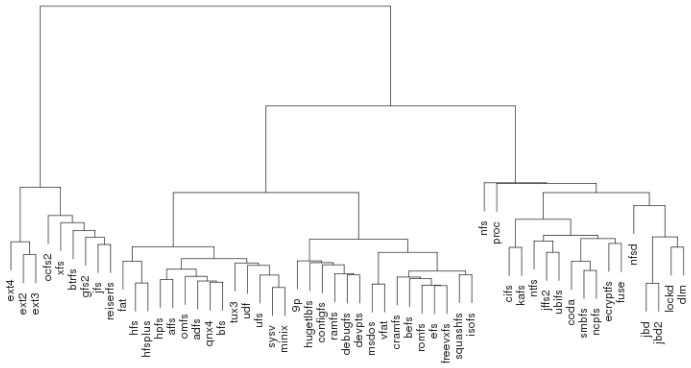
Ward's method. This method works like this: for each cluster we compute the sum of squared deviations from the cluster's centroid. Then we sum up all these sums and get a total error sum. At each step we merge the two clusters which minimize the increase of total error sum.

McQuitty's method. In this method, after each merge, the distance between the new cluster and the old ones are computed based on the distances of the two clusters that were merged. In the example below, two clusters, A and B, were merge and formed a new cluster E. The distance D3 between E and another cluster C is defined to be D3= (size(A)×D1 + size(B)×D2)/(size(A) + size(B)). This method is also call WPGMA (Weighted Pair Group Method with Arithmetic Mean).



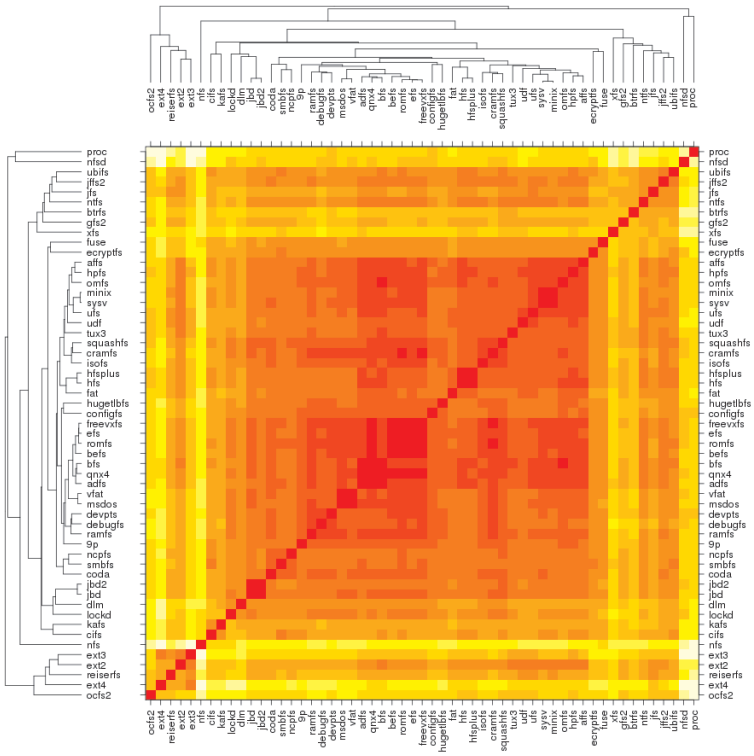
Clustering using the Hamming distance and Ward's method. Let's now take a look at how the heatmap of the Hamming distances we introduced earlier looks like when we reorder it using Ward's algorithm. The most noticeable thing is the distinctive division of the map the due to nfs/proc and nfsd. Many things are clustered in an expected ways: the ext2/ext3/ext4 family (lower left corner), the coda/smbfs/ncpfs network file systems, the jbd/jbd2 journaling services.

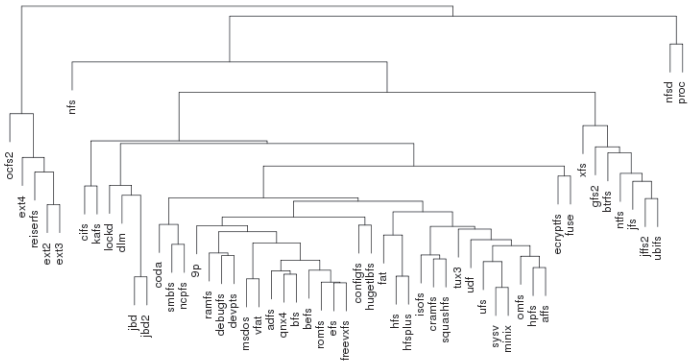




Dendrogram of the clustering using Hamming distance and Ward's method. If we look only at the dendrogram we can notice that this method divided the file systems in two big parts: complex disk-based systems including cluster file systems (which contains a disk-based part inside them), and everything else. We can also observe that most of the ancient file system category is contained almost completely in one big branch (fat–minix). Some unusual matches: 9p is situated quite far from the rest of the network file systems and tux3 ends up keeping company to the group of ancient file systems. In contrast, btrfs enjoys the neighborhood of xfs and gfs2 in the upper class of complex file systems.

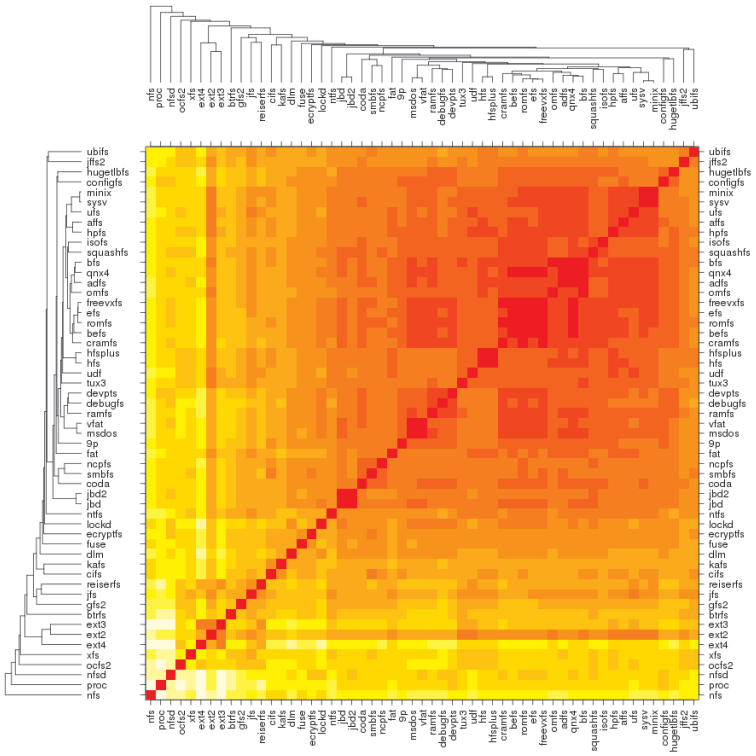
Clustering using the Hamming distance and complete linkage. Unlike the previous method, the rearrangement using the clustering based on furthest neighbor strategy (also known as complete linkage) creates a heatmap with a nicely defined center. The kernel is formed by a group of ancient file system and a few memory-based file systems. In the lower left we can also notice a close-knit group formed by ext2/ext3/ext4, ocfs2 and reiserfs. As before, a few classic modules, jbd/jbd2 and coda/smbfs/ncpfs, are again together.

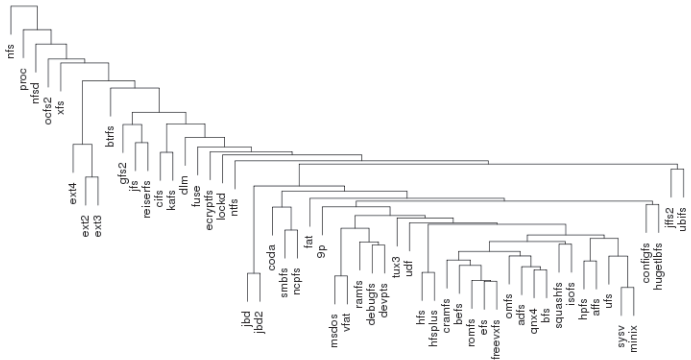




Dendrogram of the clustering using the Hamming distance and complete linkage. Here we can see that the big nice split from Ward's method is replaced by a more scatter division. The complex disk-based filesystem are now split in two parts, ocfs2–ext3 and xfs–ubifs, the second of them being muddle by two flash-based file systems (jffs2 and ubifs). btrfs is again next to gfs2 and close to xfs but, surprise, also next to ntfs. Like before, tux3 is close to a bunch of ancient file systems. isofs, squashfs and cramfs, two read-only file systems are together from the start this time with romfs, the only other read-only file system, keeping a decent distance from them.

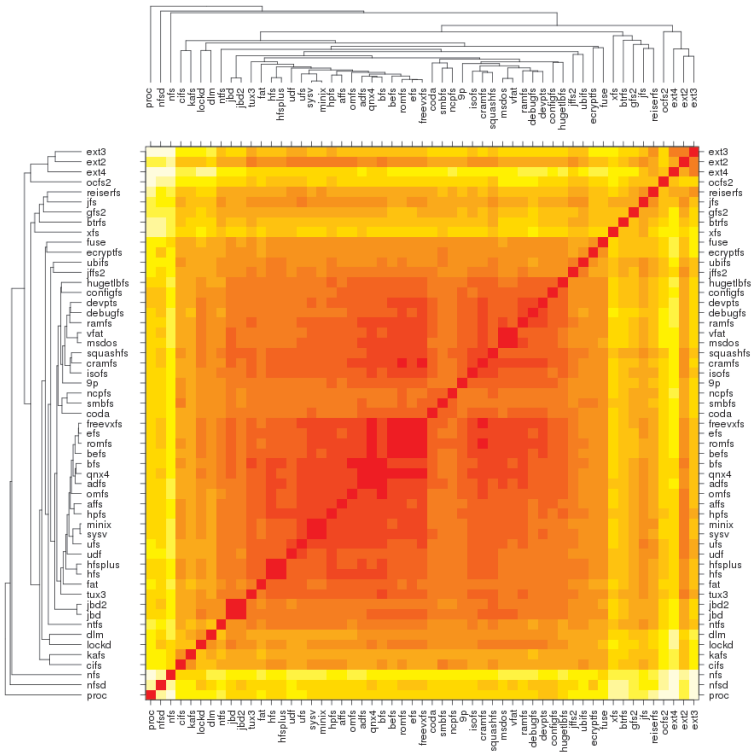
Clustering using the Hamming distance and group average. The group average method, usually used to identify bell-shaped clusters, generates a heatmap with a prominent off-center kernel. As in the previous case, this is made up of mostly by ancient file systems. Easily noticeable groups are again formed by the ext2/ext3/ext4, jbd/jbd2 and coda/smbfs/ncpfs.

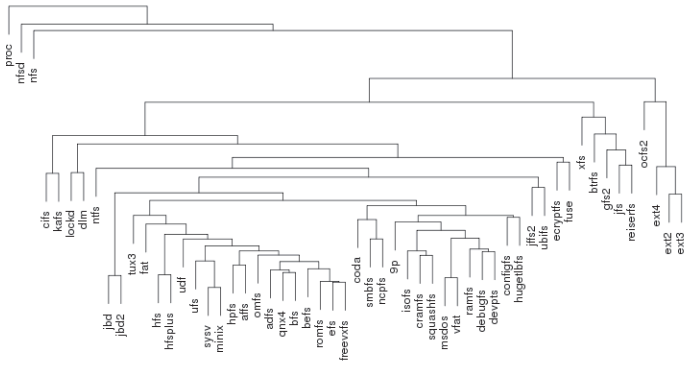




Dendrogram of the clustering using the Hamming distance and group average. In this representation, the skew is also very visible. From a high-level perspective, we have a big cluster jbd–ubifs and a few small other ones (ext4–ext3, gfs2–reiserfs and cifs–kafs) which are merge in the final merging steps with some file systems (nfs, btrfs, dlm, etc). The intuition behind this is that, as the file systems use more and more external symbols, they become more loosely connected.

Clustering using the Hamming distance and McQuitty's method. The heatmap in this case is somehow similar with the one from complete linkage.

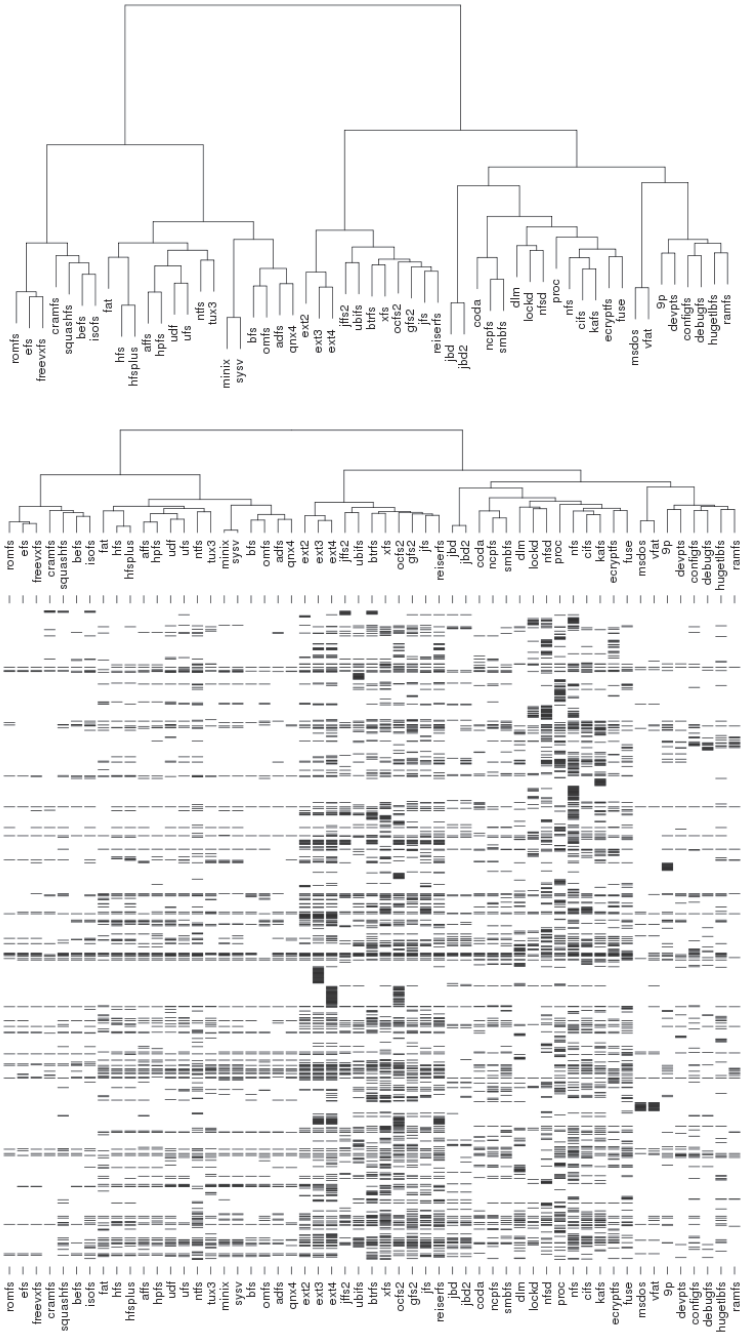




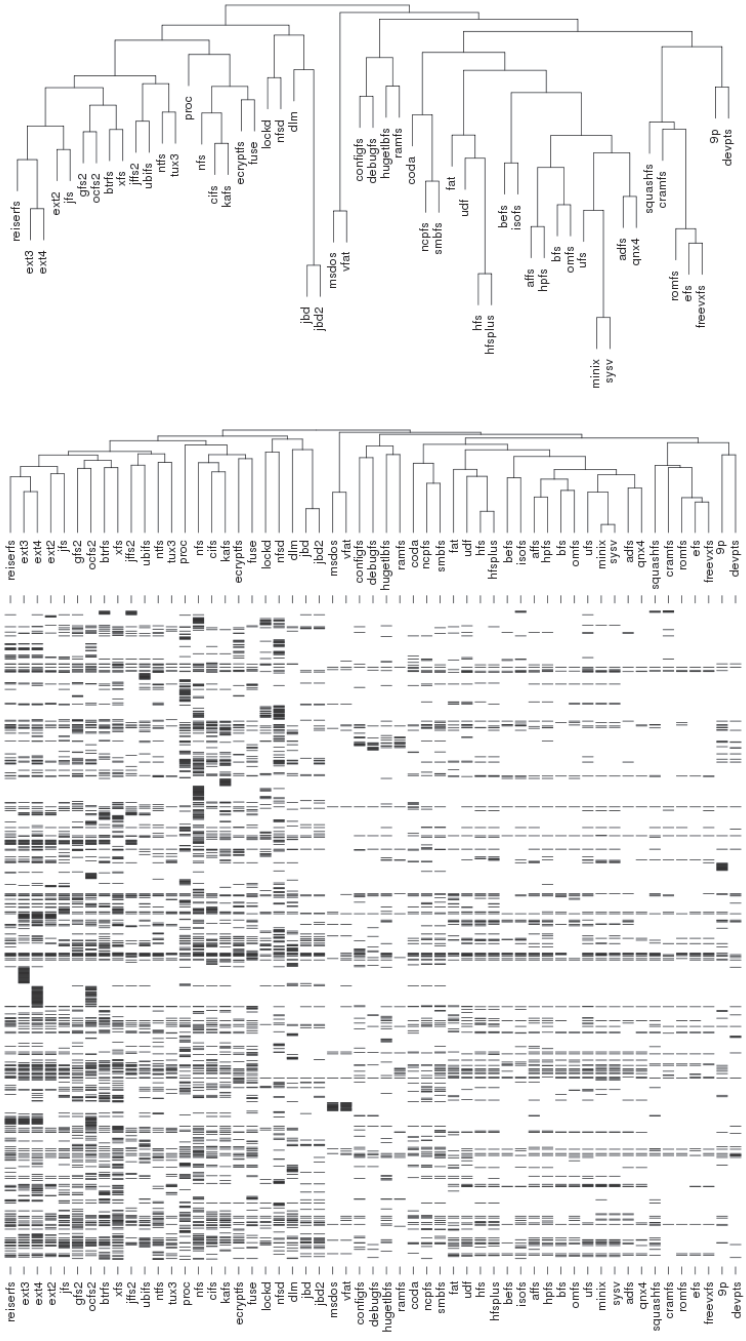
Dendrogram of the clustering using the Hamming distance and McQuitty's method. If we ignore the skew induced by proc/nfsd/nfs and the fact that the complex disk-based file systems (xfs–reiserfs and ocfs2–ext3) are not sharing the same level, then the resulting tree is nicely split in similar things.

Following are a set of dendrograms using Canberra distance. In our case, this metric is equivalent with the number of different external symbols between two modules. After each dendrogram a reordered map of symbols is also plotted.

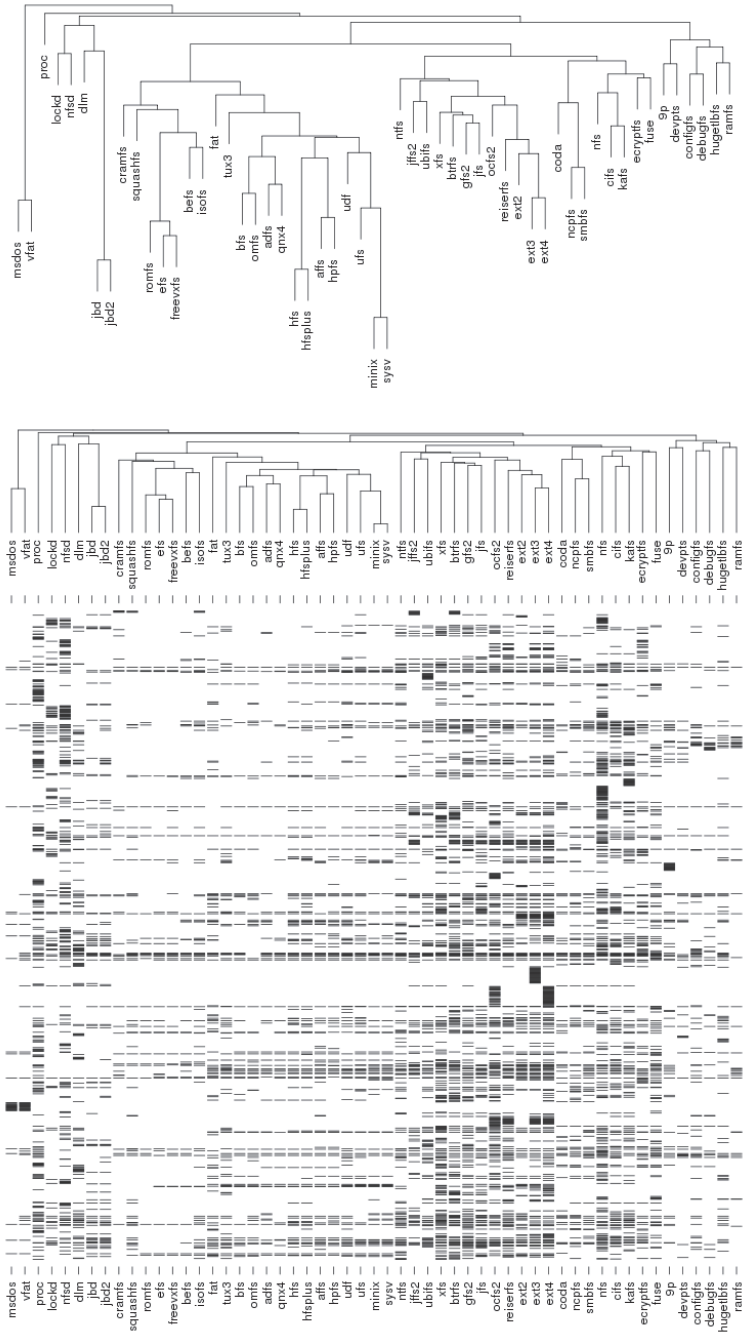
Clustering using the Canberra distance and Ward's method.



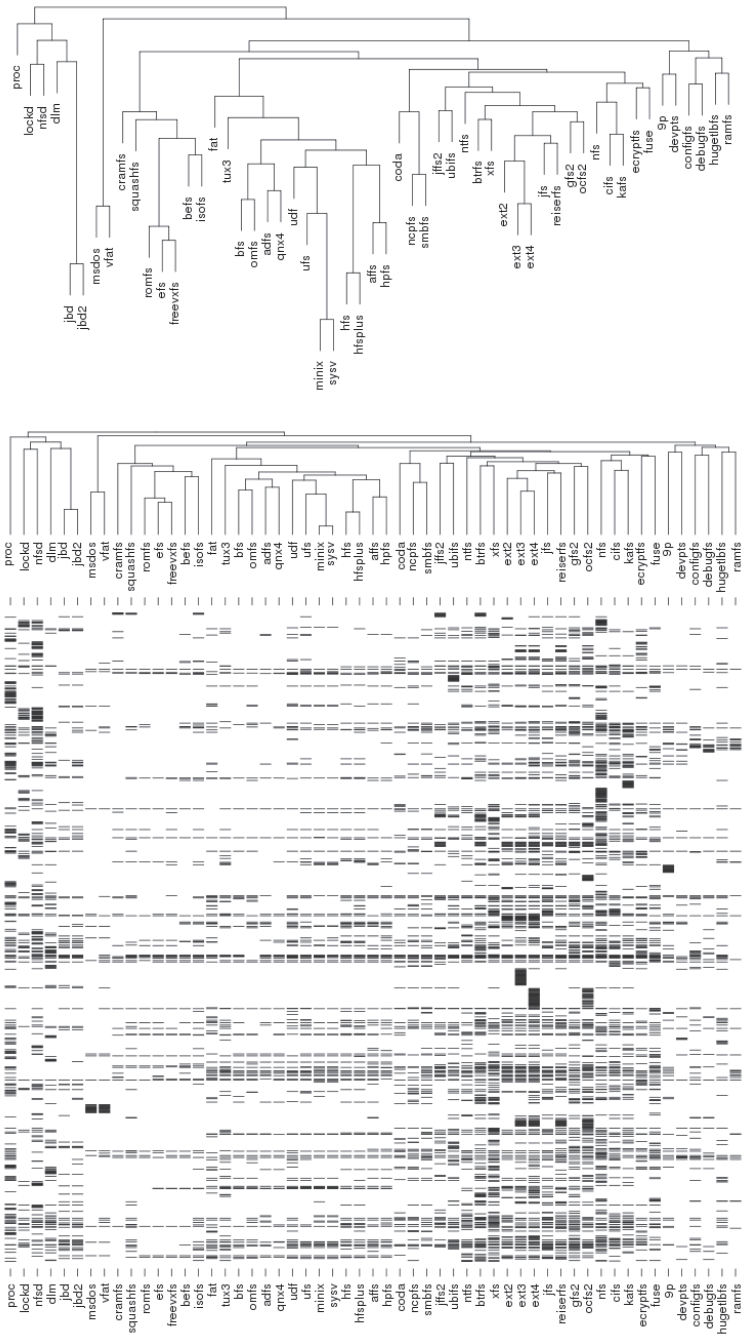
Clustering using the Canberra distance and complete linkage.



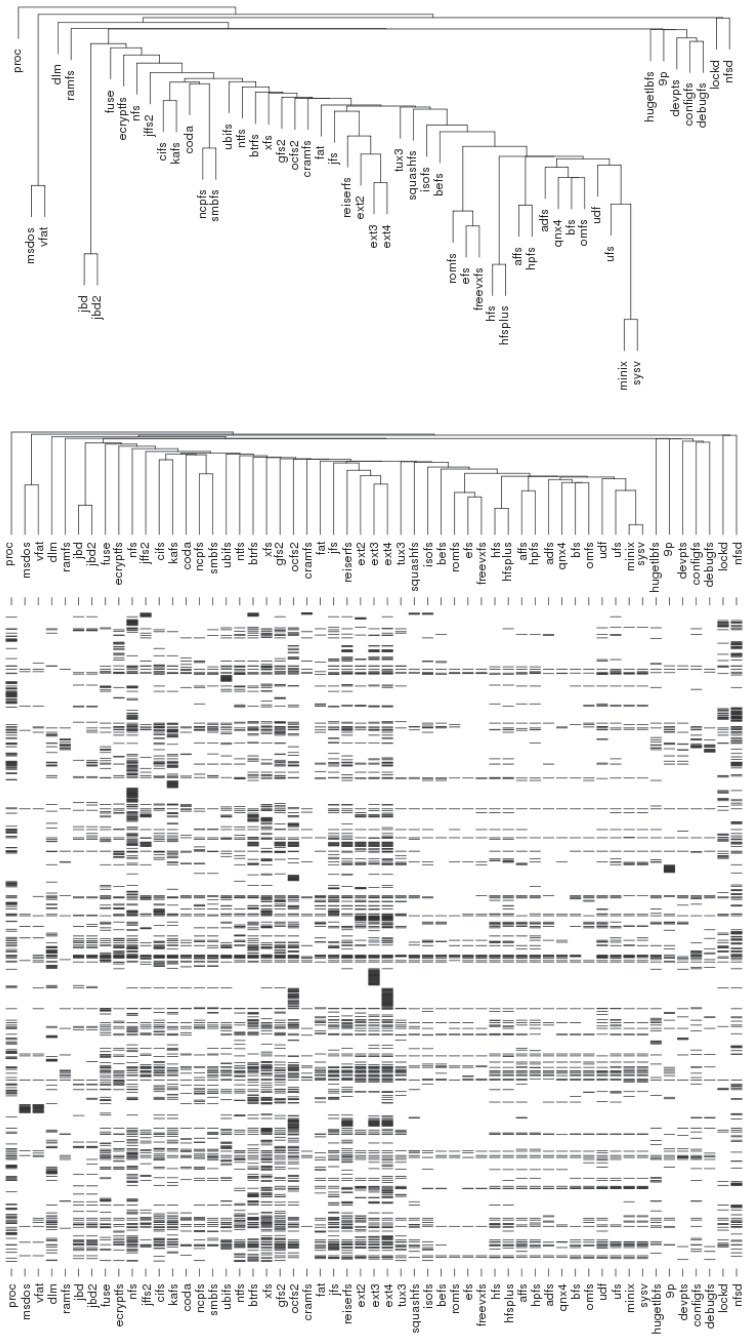
Clustering using the Canberra distance and group average.



Clustering using Canberra distance and McQuitty's method.



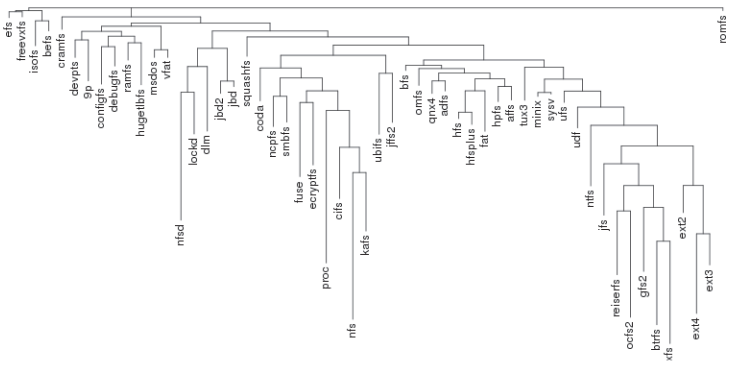
Clustering using Canberra distance and single linkage.

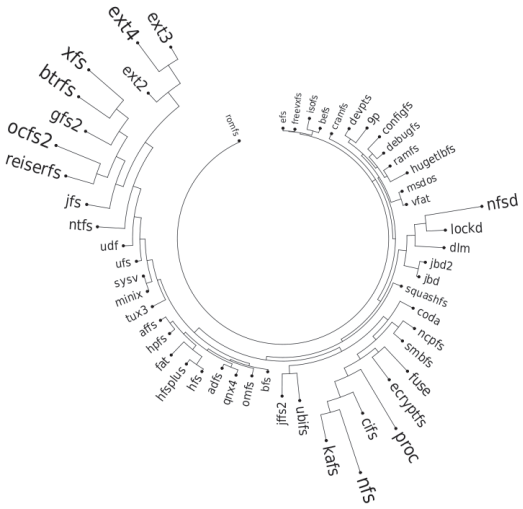


Phylogentic tree. Pars is a program that can construct an evolutionary tree which requires a minimum number of changes (maximum parsimony). The program is part is part of PHYLIP, a computational phylogenetics package created and maintain by Joseph Felsenstein.

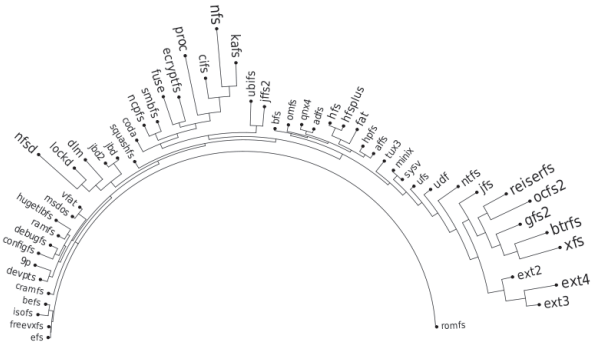
The input to Pars is number of species, each described using a string of characters. Usually each character is either "0" or "1" indicating the presence of absences of a certain feature but Pars also capable of dealing with up to 8 states plus "?" which indicates an unknown state. In our case we only need "0" and "1" and each position in the string encodes a certain external call.

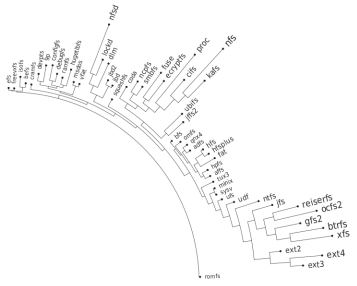
The result is the tree from upper right. It looks like a dendrogram but it is slightly different. This time the length of any vertical line is proportional with the number of changes between two states. We can see for example that msdos and vfat are both very close to the their parent while ext4 and ext3 are much farther apart.

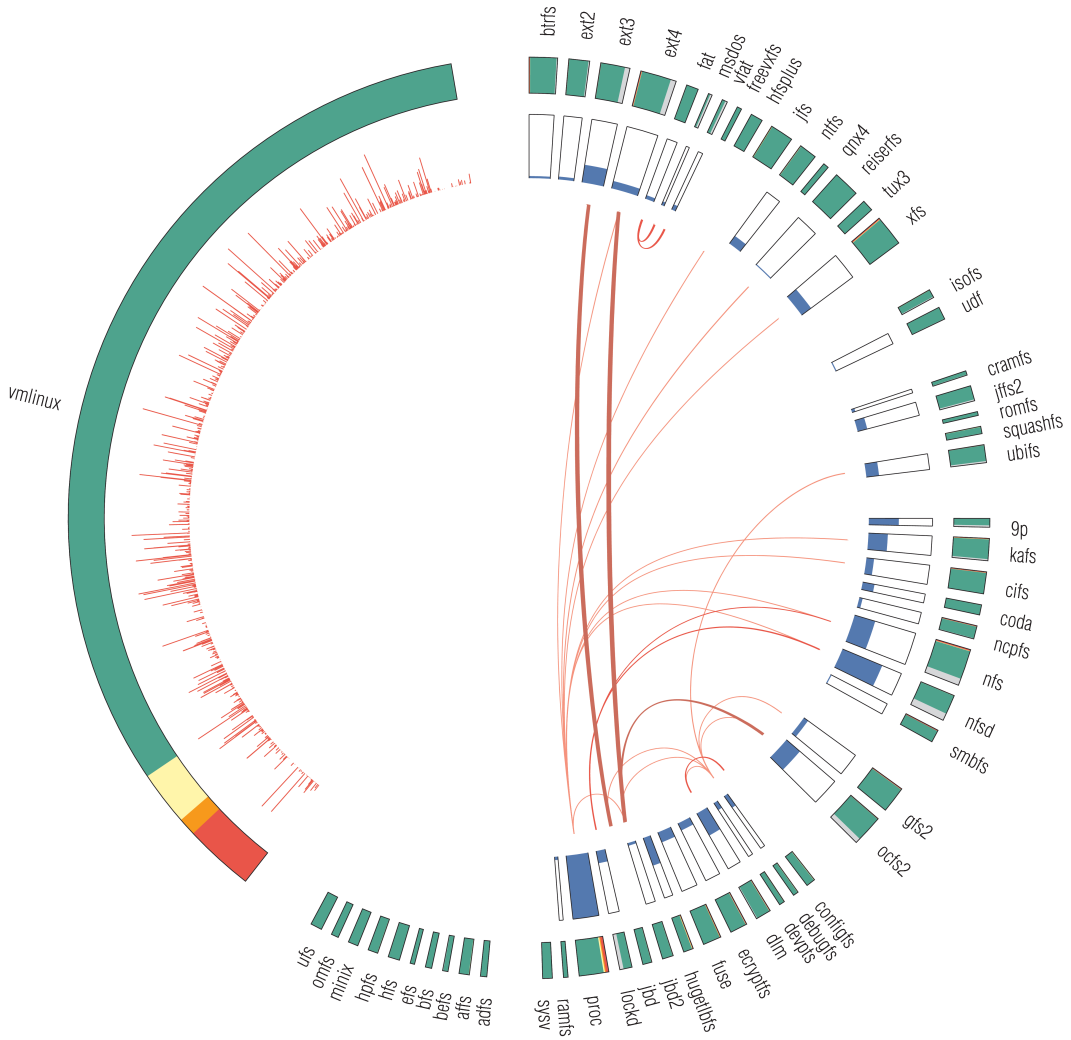




Circular representations of the phylogentic tree. This are three alternative representations of the same tree. The size of the text is proportional with the depth.







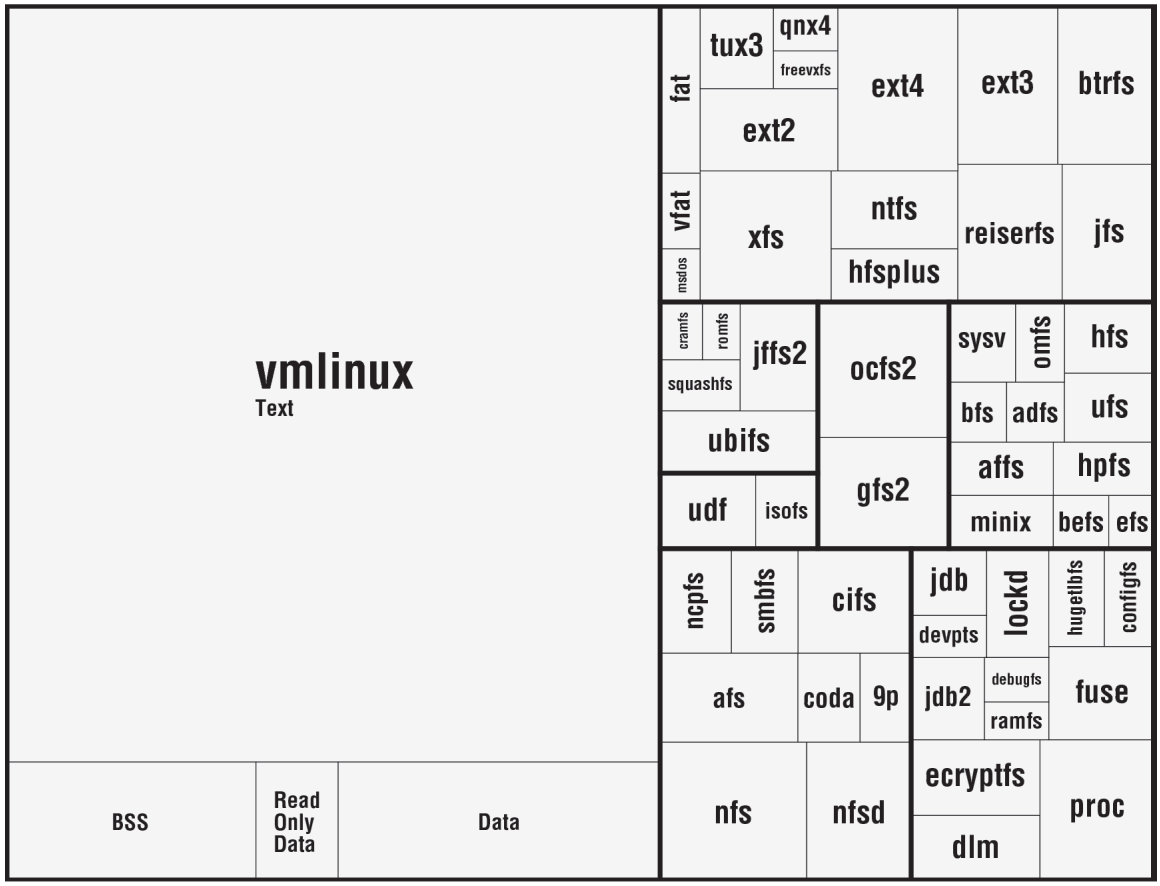
Circos. Circos is a visualization tool by Martin Krzywinski. The initial purpose was to provide a better representation of various genomic data but the program was successfully use to produce very nice graphical representations of other type of data.

Let's now look at our plot. On the outer edge we have the file systems split in categories (from top to bottom: disk-based, optical mediums, flash-based, network-based, cluster-based, memory-based, ancient). The size is proportional with the number of external symbols. Colors indicate the type of external symbols. The green represents functions, red is data, orange are read-only data, and light yellow is uninitialized data (BSS). To give a sense of proportions, the external symbols exported by the Linux Kernel, vmlinux, are also depicted. It was compiled in the same configuration as the rest of the file systems. To give some numbers: it exports a total of 9310 external symbols out of which 8047 are functions, 621 are writable variables, 159 are read-only and 483 are BSS data. The gray area from file systems indicates the external symbols which are not satisfied by the kernel but by some other kernel module. This is noticeable for nfs, nfsd and also the users of jbd/jbd2: ext3, ext4, ocfs2.

On the inner edge of vmlinux there is a plot that indicates the frequency which which each exported symbol is used by the file systems from right. One thing we noticed here is that variables are used pretty much with the same frequency as the functions.

The set of boxes from the inner edge of the file systems represents the percentage of the external symbols which are unique to each file system. We can see that virtually all the external symbols used by proc are only used by him. But having unique external symbols is not a rare feature: with the exception of ancient file systems all the other categories have members with various degree of "uniqueness".

The red arcs from inside depicts the use-provide relationships between the file systems. As expected, the memory-based modules are the ones that are the main providers with proc and debufgs being the most popular one. We can also see that lockd is used by nfs and nfsd and also the relation between fat and vfat/msdos. A notable thing: there is no link between dlm and ocfs2/gfs2 because only the main kernel module was considered.



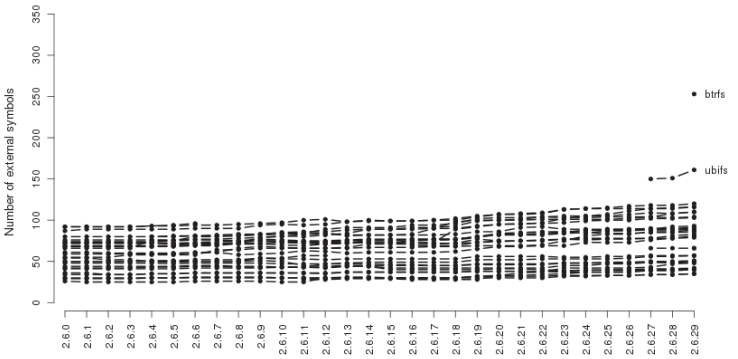
Treemap. A treemap is a visual representation of hierarchies using nested rectangles. The first version was introduces by Ben Shneiderman in 1990 in the context of visualization of directories structures. In the representation we have below the size of the rectangle for each file systems is proportional with the number of external symbols used by it. The symbols exported by vmlinux are also depicted. The thicker lines indicate the boundaries of the seven categories we introduced earlier.

3. Linux Kernel 2.6.x

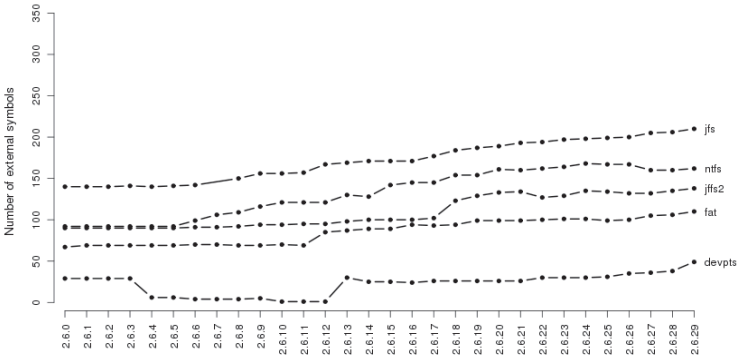
In this section we are going to look at the relations between 1377 file systems compiled from Linux Kernel 2.6.0–2.6.29. More details about how I compiled them can be found in Appendix A. Note that this time the final 2.6.29 is used. This means that tux3 is missing.

Like the previous one, this section will also consists of a sequence of commented figures.

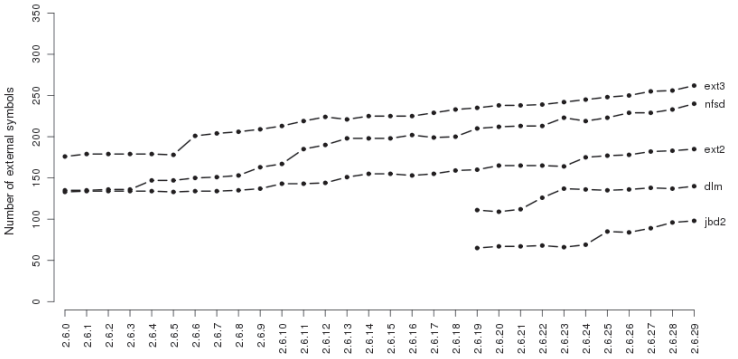
Number of external symbols over time. In order to avoid clutter, this graph is split in 6 parts. The split was done based on how much external symbols file systems gain over the whole span of their life. The first plot contains a big group of 35 file systems that shows only very little changes. Beside btrfs and ubifs, which stands out due to their big number ob external symbols, the other 33 are the following: adfs, affs, befs, bfs, coda, configfs, cramfs, debugfs, devfs, efs, freevxfs, hfs, hfsplus, hpfs, hugetlbfs, intermezzo, isofs, jbd, jffs, lockd, minix, msdos, ncpfs, omfs, qnx4, ramfs, romfs, smbfs, squashfs, sysv, udf, ufs, vfat.



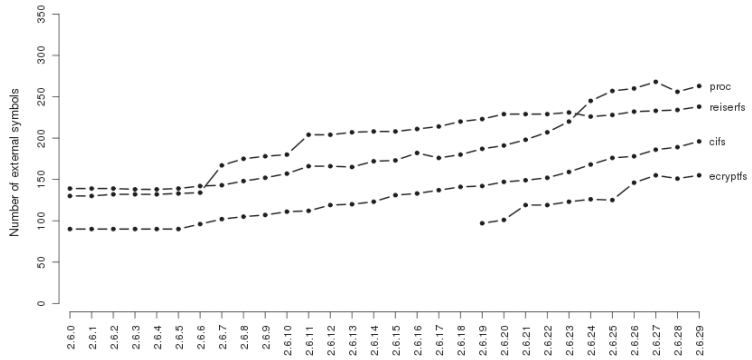
Only two notable things here: the race between ntfs and jffs2 which start with a microscopic distance of only 2 symbols in 2.6.0, go apart to a distance of 43 symbols in 2.6.17 and end up with a distance of 24 in 2.6.29, and the sudden sink of devpts in 2.6.4 which is followed by a a similar increase in 2.6.13.



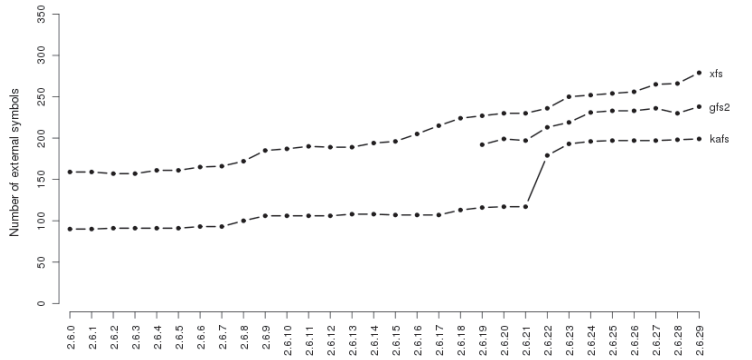
Another close race is going on now between nfsd and ext2. They also start at a distance of 2 symbols and the distance keep growing ending up at 55 symbols in 2.6.29.



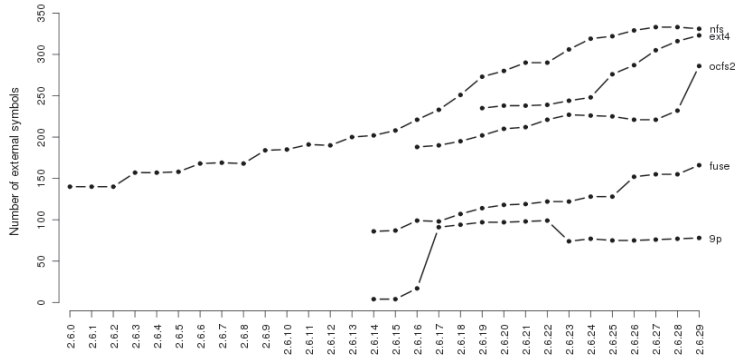
Yet another interesting race is featuring this time proc and reiserfs. proc starts with an advantage of 9 symbols but reiserfs is taking the lead in 2.6.7 and surrenders in 2.6.24. By 2.6.29 the distance gets to 25 symbols.

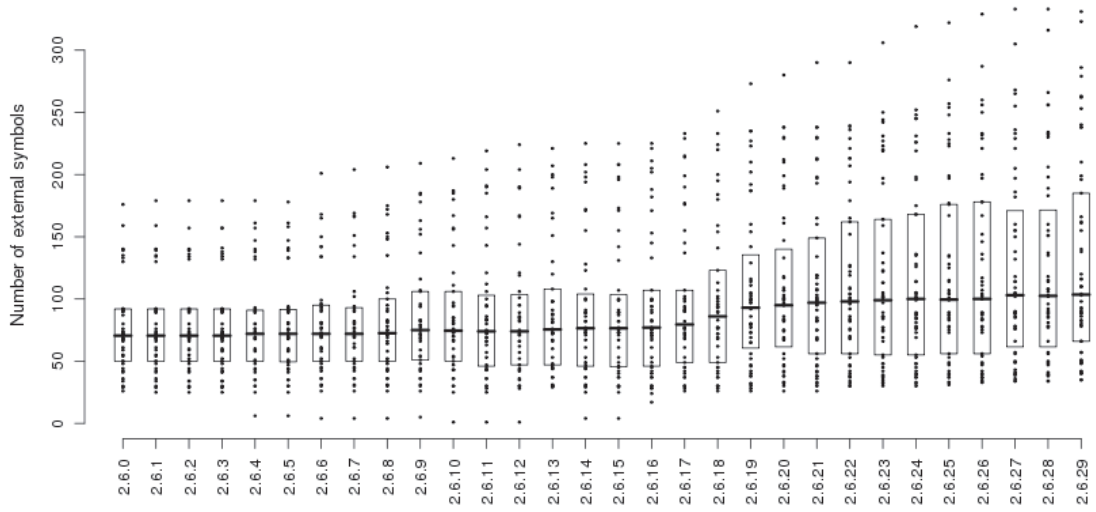


The only remarkable thing is the impressive jump of 62 symbols kafs, the Andrew File System, is making in 2.6.22. As we'll see a little later, this will earn him the second place in the top of biggest jumps.

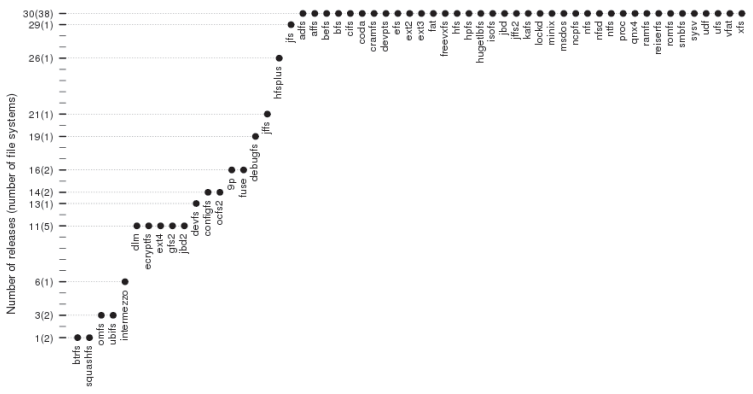


In here we have 9p the absolute winner of the biggest increase (74 symbols in 2.6.27) and also the biggest decrease (25 symbols in 2.6.23). The 54 symbols gained by osfs2 in 2.6.29 put him in the third place in the biggest increase contest. Another winner is nfs which holds the absolute record for the biggest difference between the minimum and maximum number of external symbols.



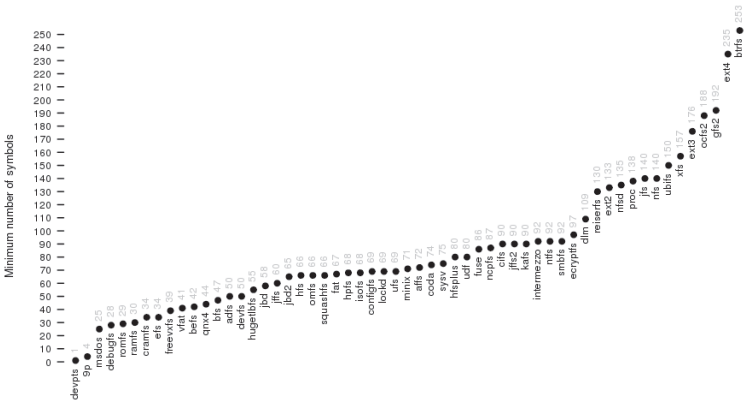


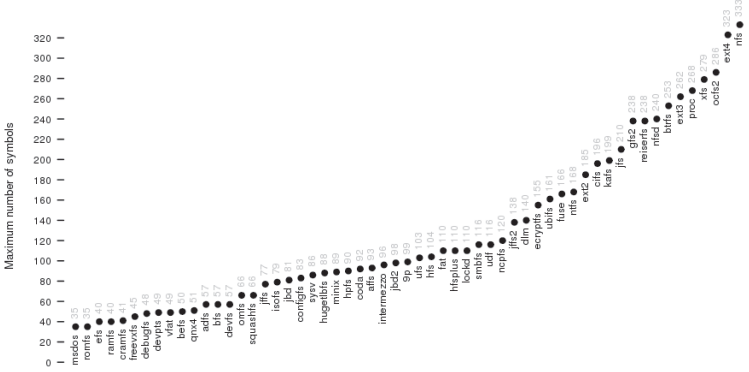
Boxplots of the external symbols for each release. In order to avoid cluttering, in this plot the lines that are show in the previous ones are omitted. The boxes indicate the interval which contains 50% of the file systems for that particular release. The thick horizontal line inside the box indicate the median. We can see that the median goes up and so does the spread of the two middle quartiles and the extremities.



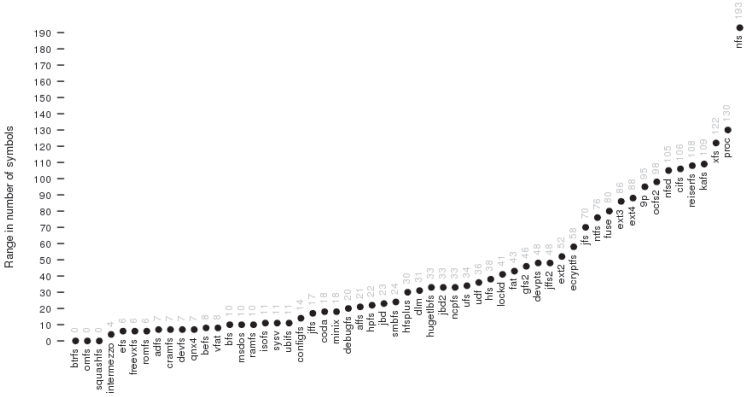
Ranking based on longevity. This is the first out of seven rankings and it shows in how many releases each of the 65 file systems show up. What is easily noticeable is that most of the file systems, 38 of them, are veterans and are present in all the 2.6.x releases. Note that there are only four disk-based systems that are no veterans: a very young one, btrfs which shows up only in one releases; a junior one, ext4 which accumulated so far 11 releases and two old seniors, hfsplus and jfs which missed the big group of veterans by only four and one releases, respectively.

Rankings based on the minimum and maximum number of symbols. The following two graphs shows the file systems sorted by the minimum and maximum number of symbols they had over their life in 2.6.x. The unbelievable number of external symbols of only one was achieved by devpts in three consecutive releases (from 2.6.10 to 2.6.12). Also hard to believe is the number of 4 external symbols scored by 9p in 2.6.14 and 2.6.15. I haven't actually test this modules so they might be broken and/or incomplete.

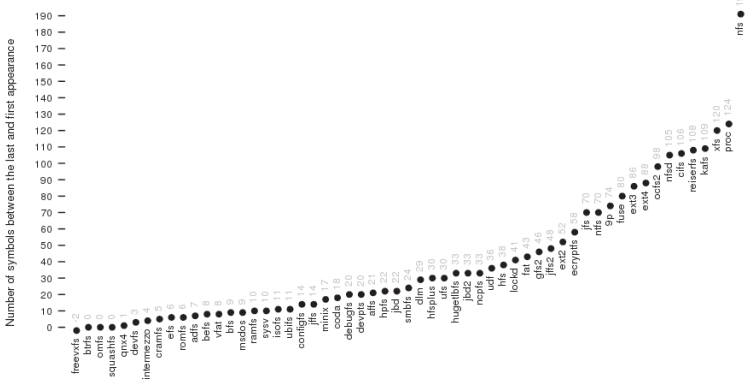




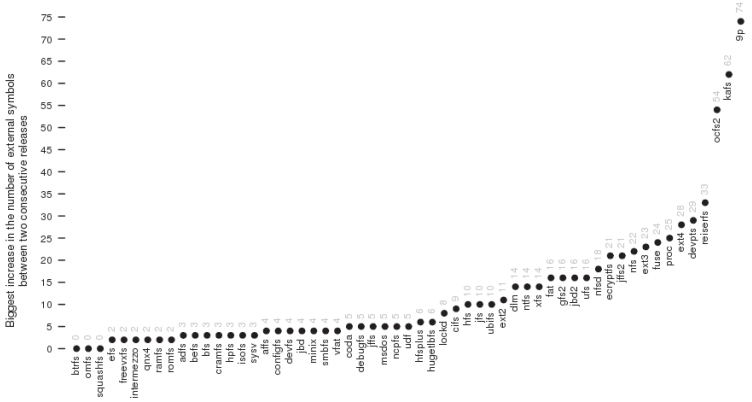
Rankings based on range or number of symbols. This plot shows the ranking based on range, the difference between the maximum and minimum number of symbols a file system had reached in its lifetime. As we pointed before, the absolute winner is nfs. Two of the file system at the other end, btrfs and squashfs might not hold their position for long considering that both only have one one presence in 2.6.x.

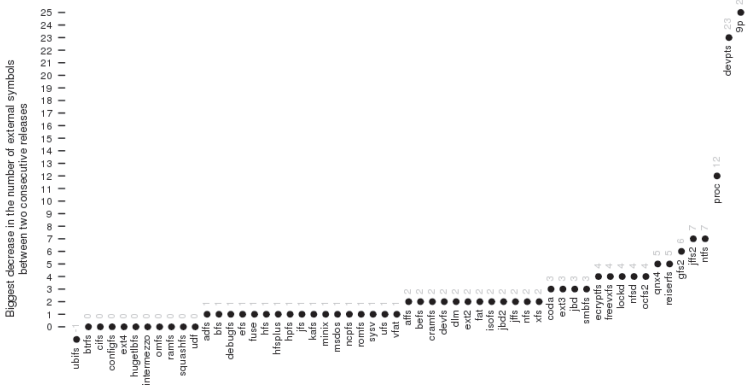


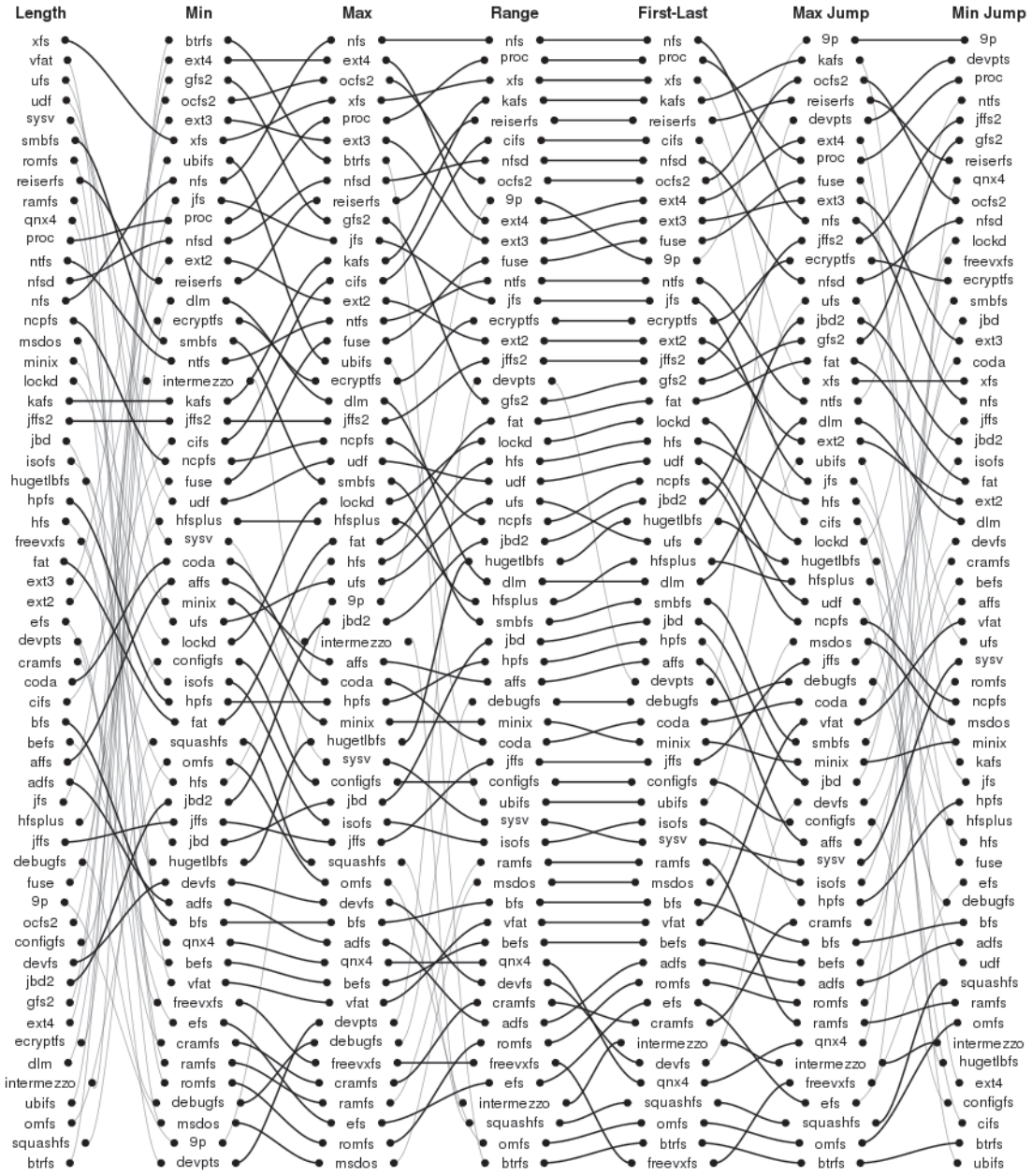
Rankings based on the difference in number of symbols between first and last appearance. This plot is very similar with the previous one. One interesting thing is that freevxfs, a veteran which didn't skip any 2.6 release, ends up in 2.6.29 with two less symbols than in 2.6.0.



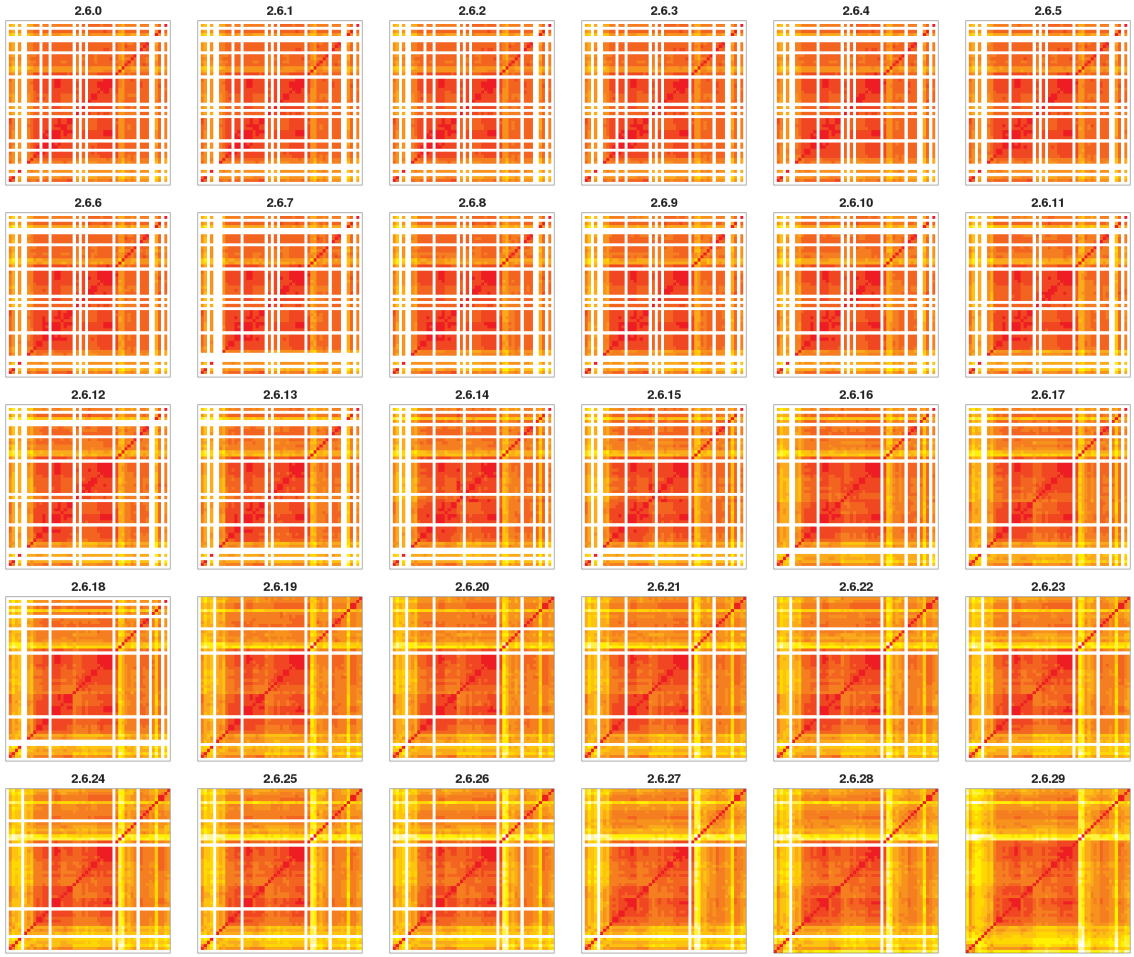
Rankings based on the biggest increase and decrease. Beside the first places which were already mentioned what can be said about this is that, despite the fact that most of the file system don't have big increases of the number of external symbols from one release to another they decrease much less than they increase. The behavior of ubifs, which never decreases, can be excused by the fact that he has only 3 presences in the 2.6.x.



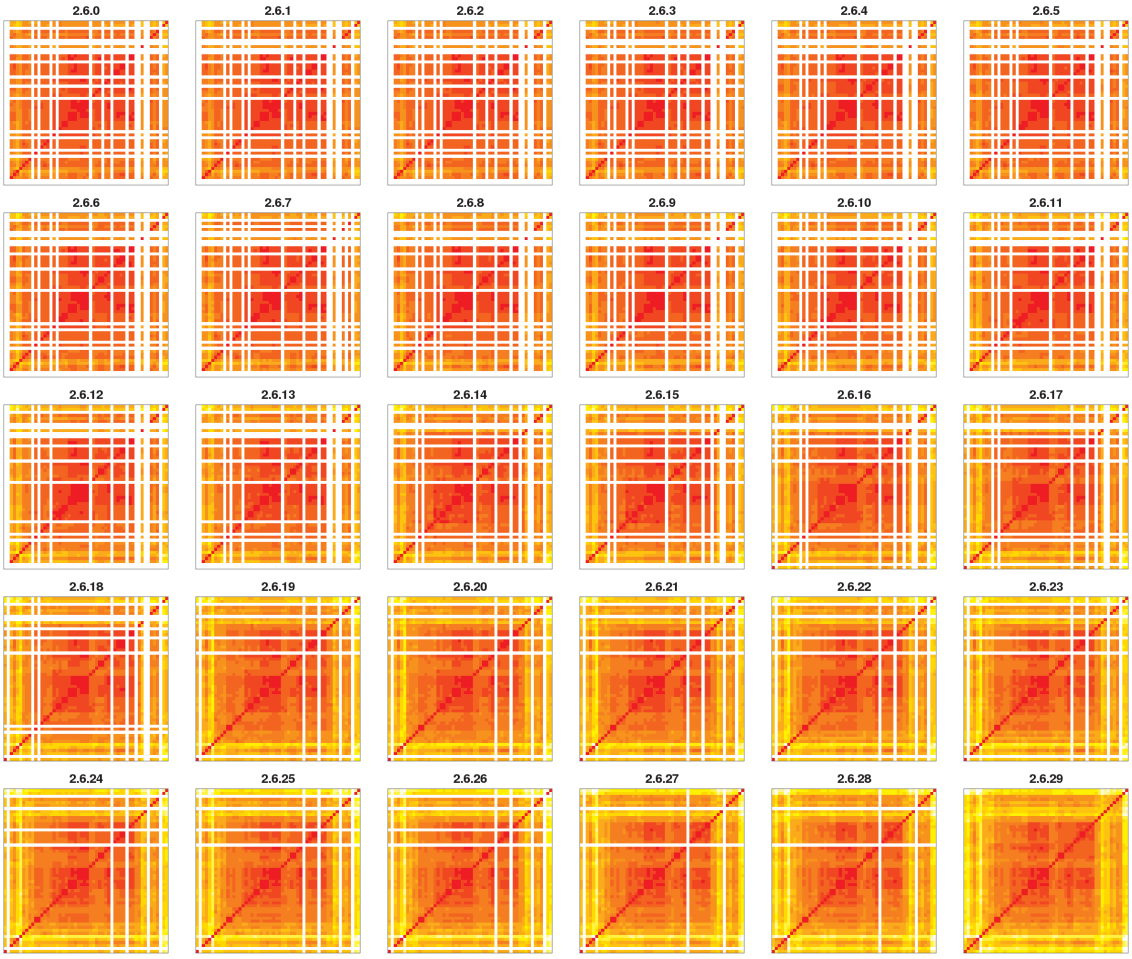




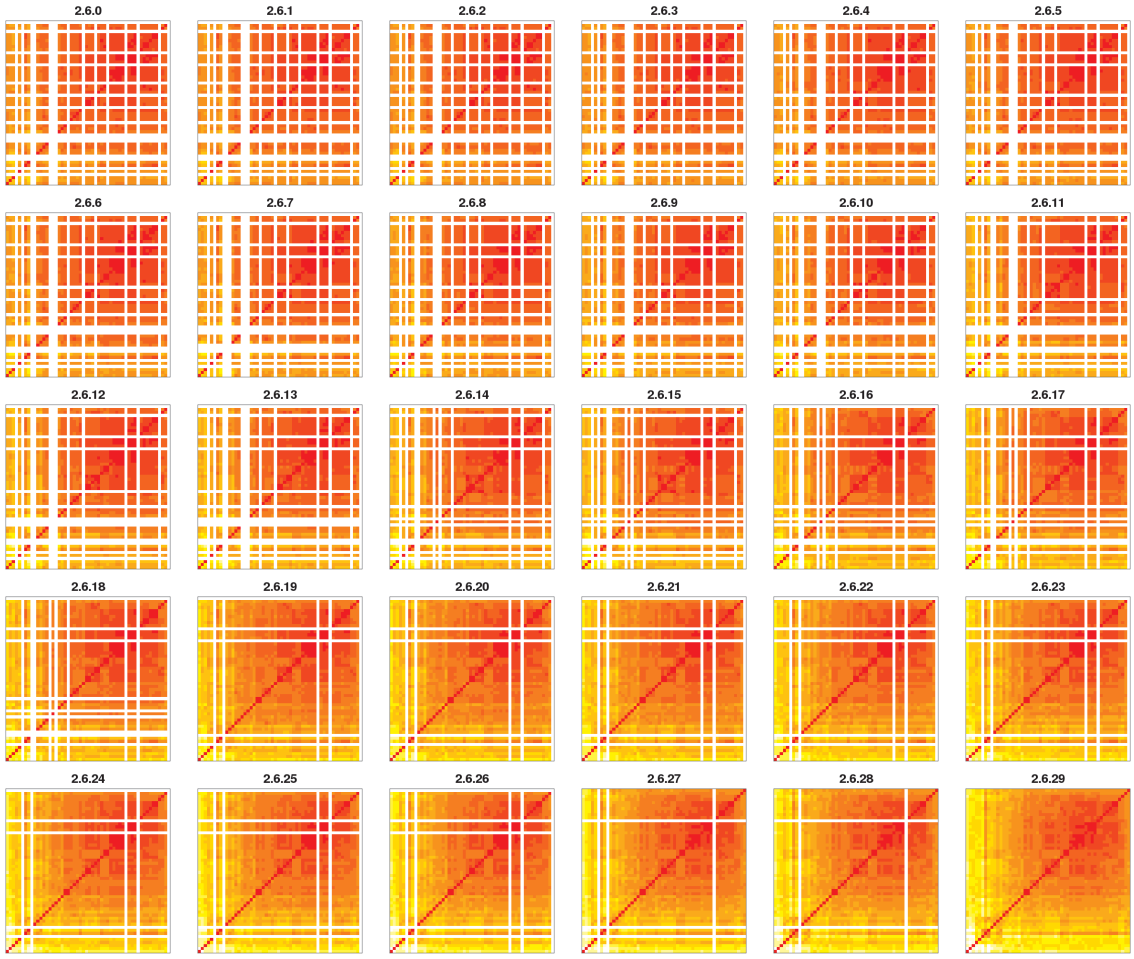
Relations between the rankings. This plot summarizes the previous seven ones. The gray lines are used when the rank changed more than 10 positions.



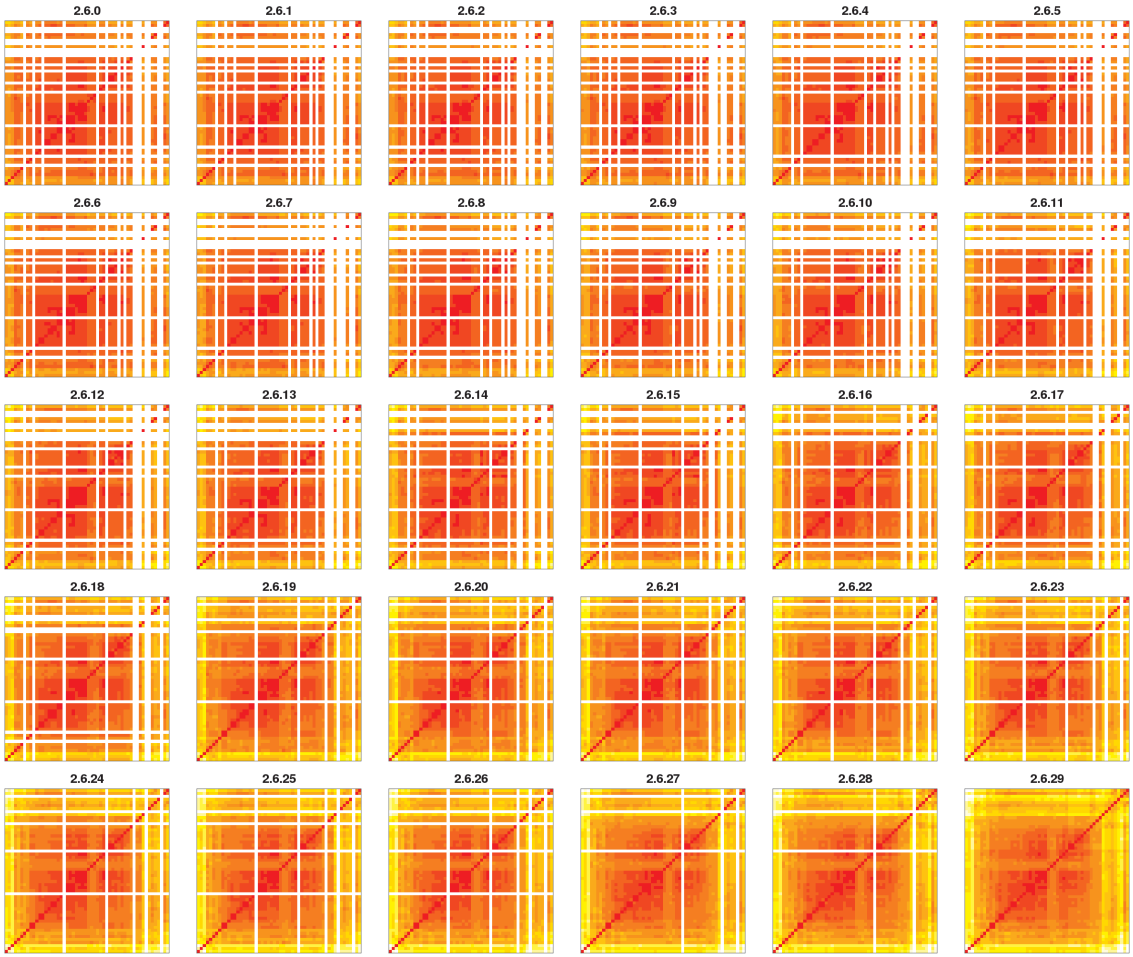
Heatmap of the clustering using Hamming distance and Ward's method. This is the first out of four animations that shows how the heatmap of the Hamming distance evolves over time. To aid the comparison, the file systems show are only the one that are still present in 2.6.29. Their position is keep fixed and is determined by the clustering done on the distances from 2.6.29. The Hamming distance is computed separately for each release. Red indicates high similarity and yellow indicates the opposite. One notable thing in all the four animations is that similarity is decreasing over time.



Heatmap of the clustering using Hamming distance and complete linkage.

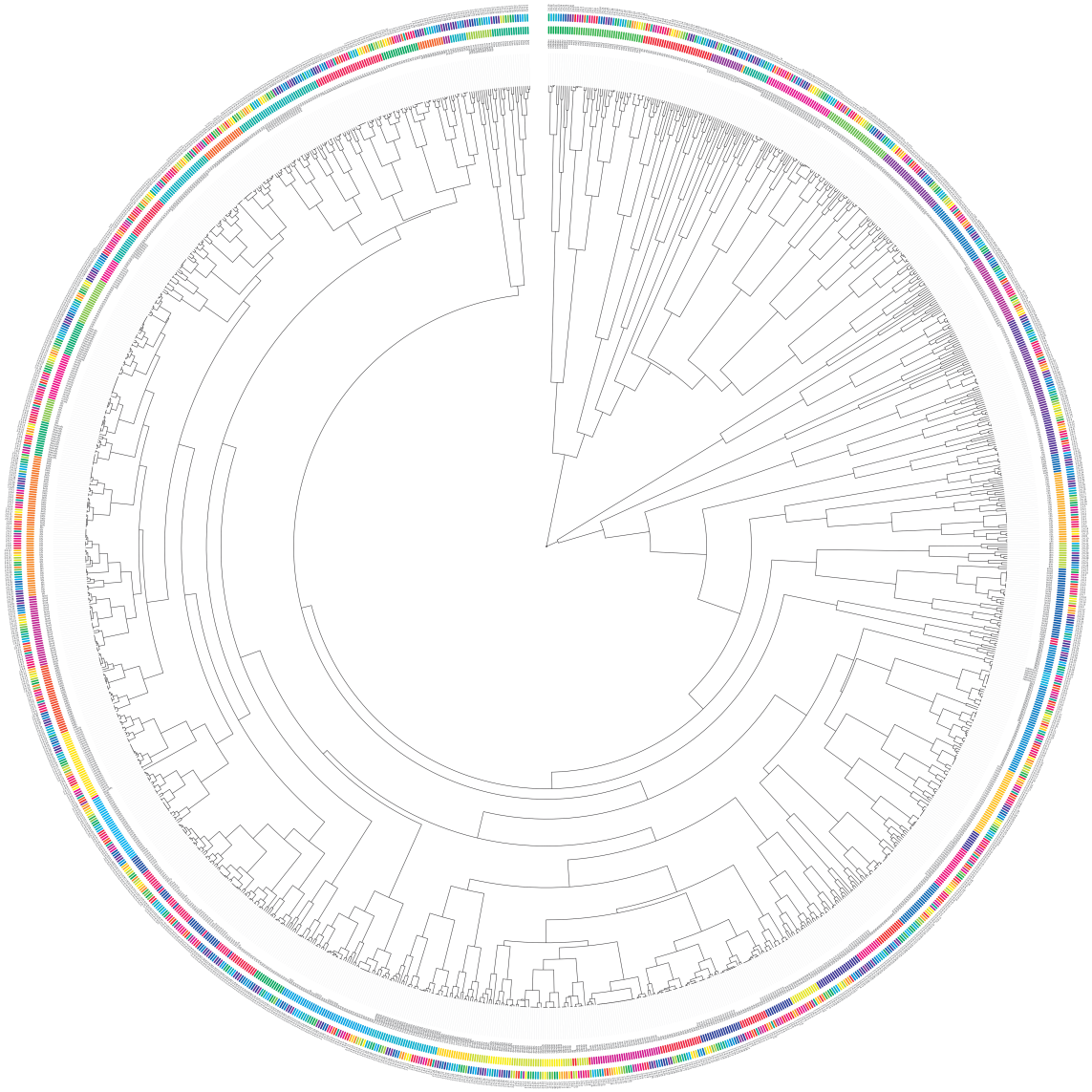


Heatmap of the clustering using Hamming distance and group average.

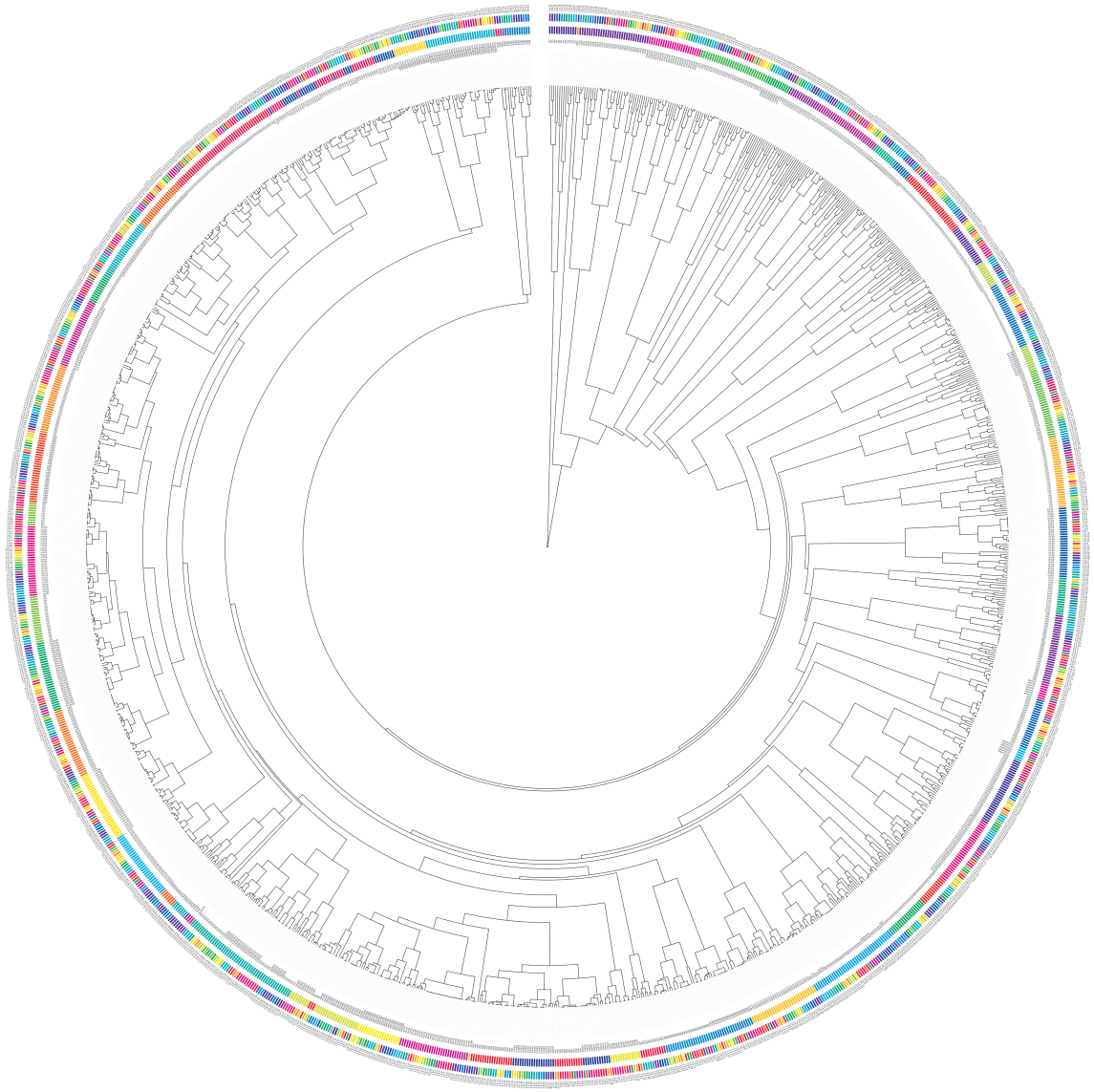


Heatmap of the clustering using Hamming distance and McQuitty's method.

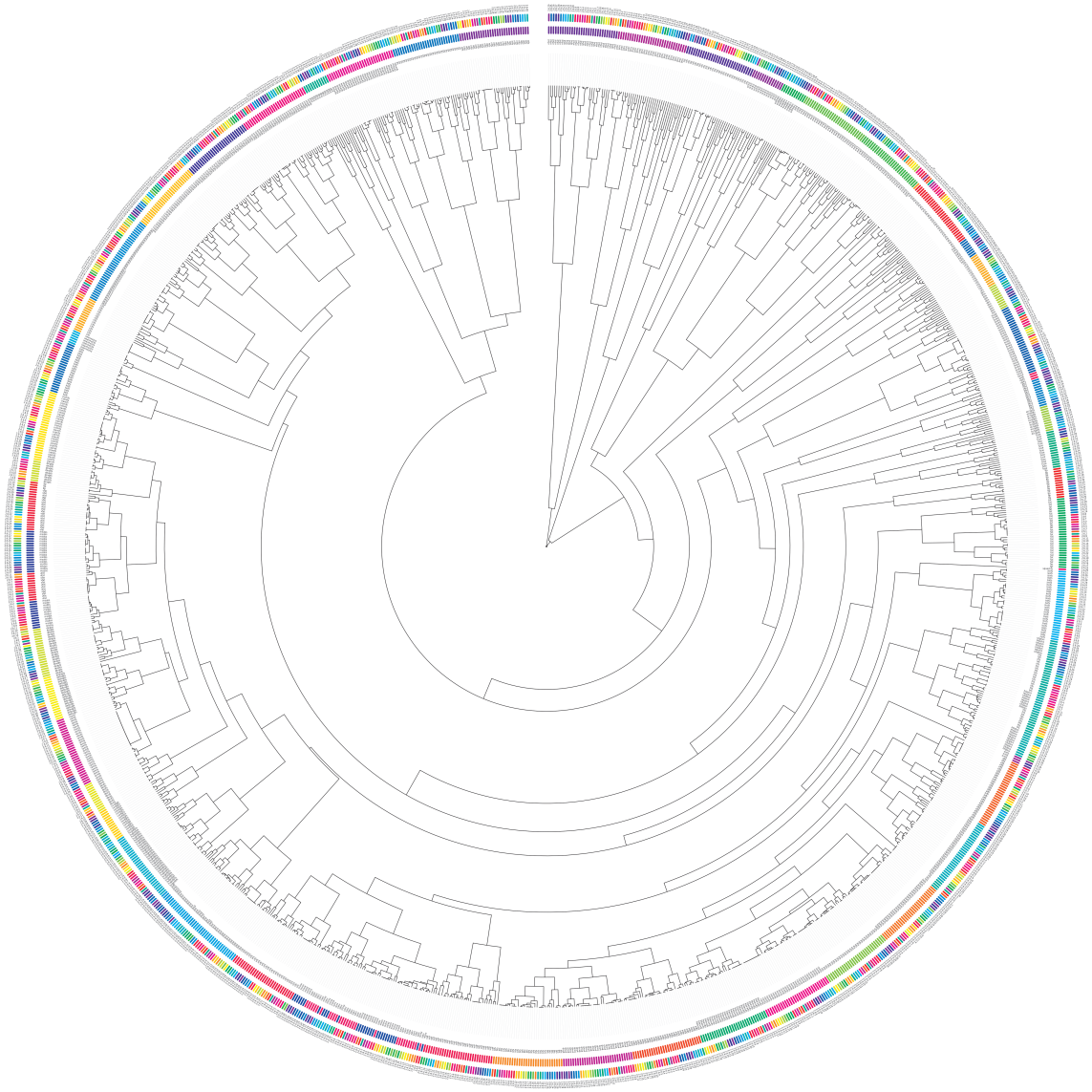
Circular dendrogram of the clustering using Hamming distance and Ward's method. This is the first out of four circular dendrograms that shows the result of clustering using the Hamming distance over all the file systems. The ticks that make up the inner ring encodes file system while the outside one encodes the version number. A high resolution image with text annotation is available upon click.



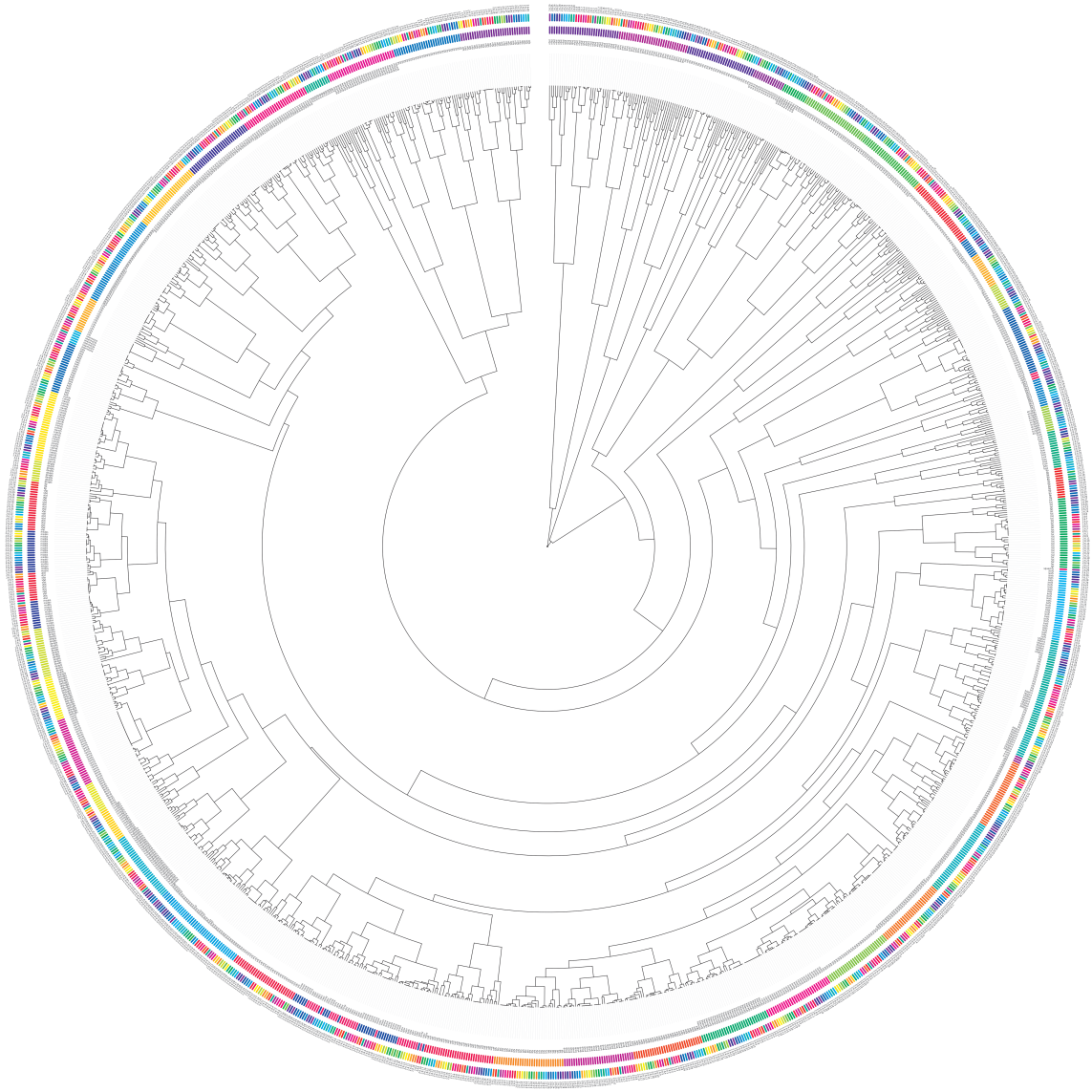
Circular dendrogram of the clustering using Hamming distance and complete linkage.



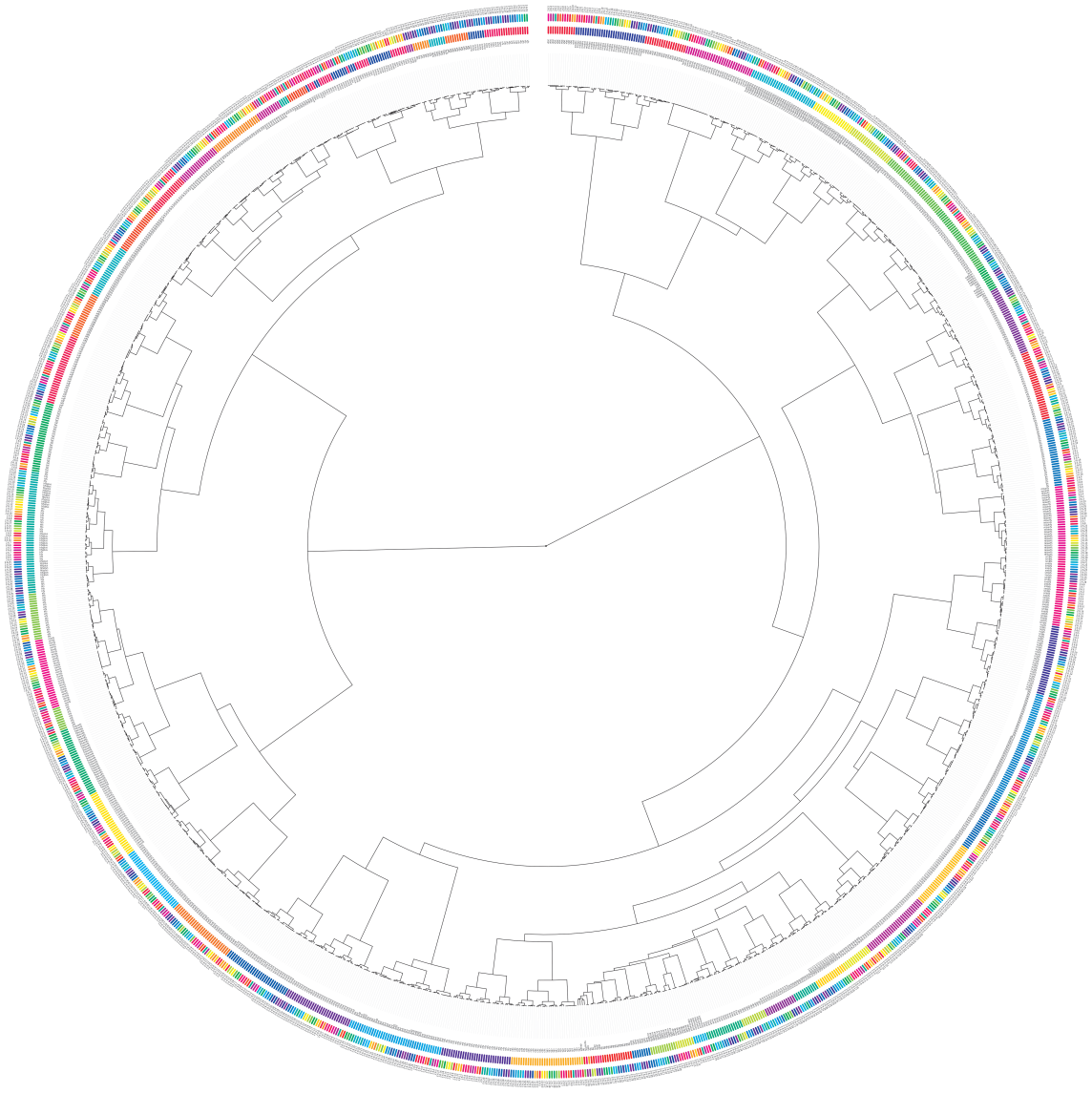
Circular dendrogram of the clustering using Hamming distance and group average.



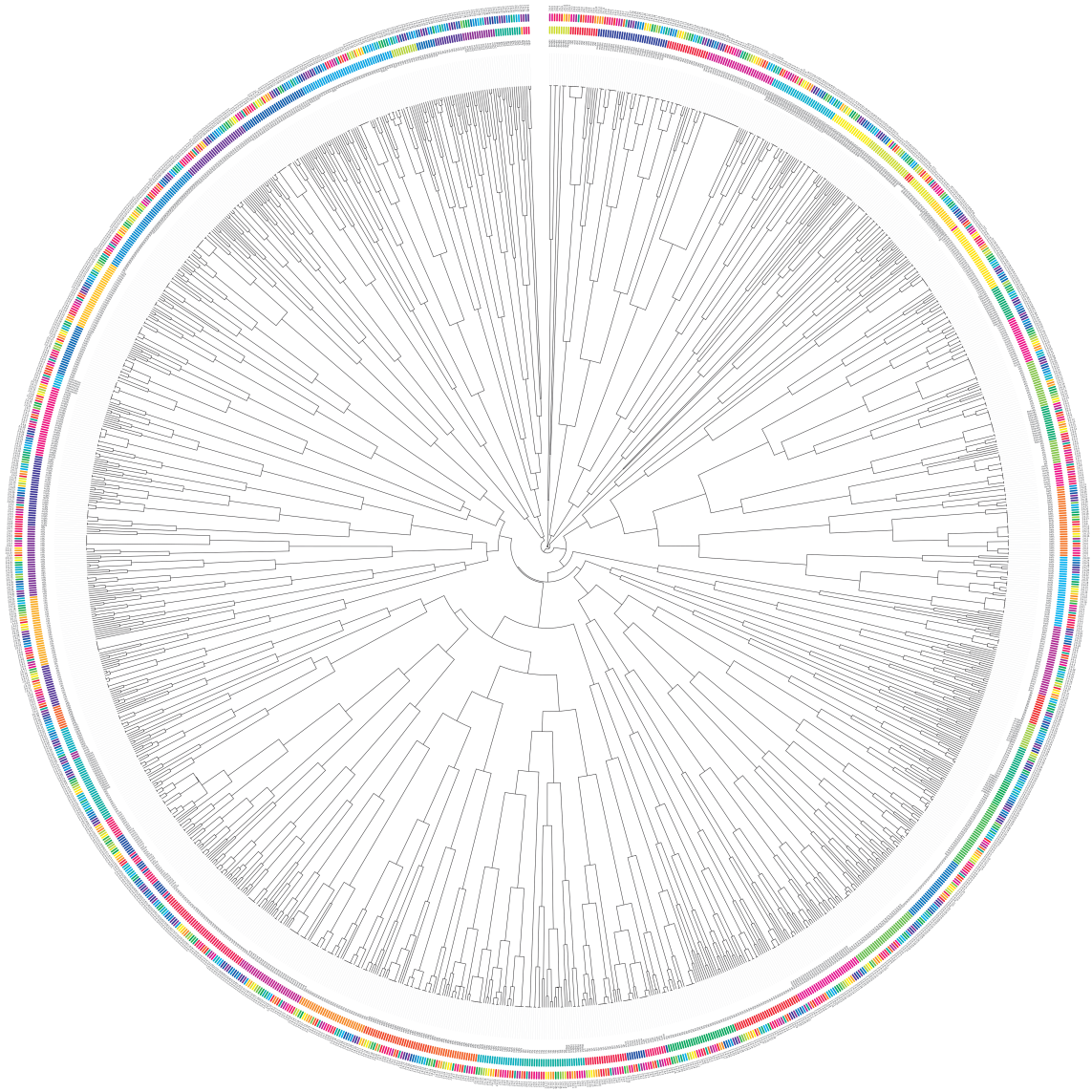
Circular dendrogram of the clustering using Hamming distance and McQuitty's method.



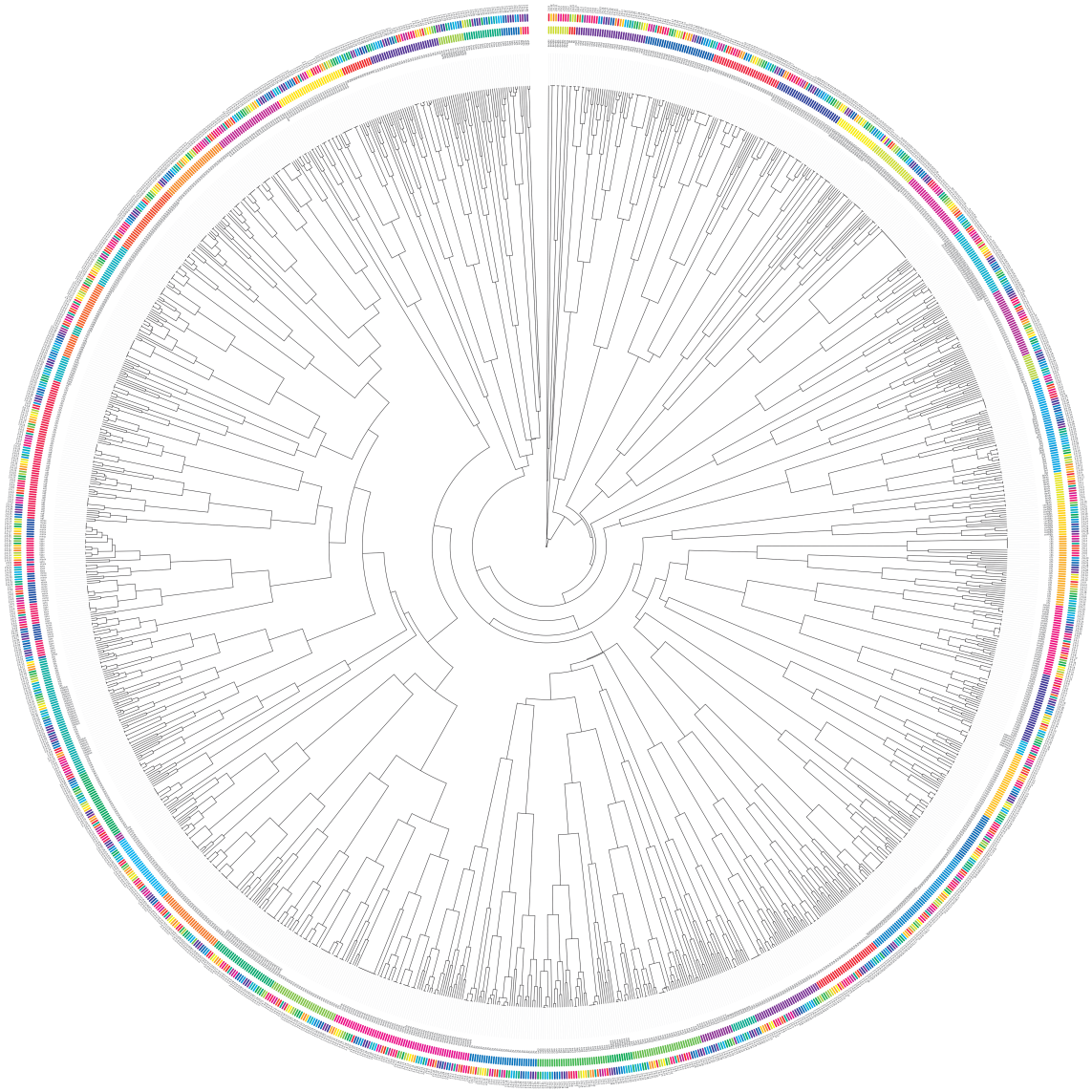
Circular dendrogram of the clustering using Hamming distance and McQuitty's method.



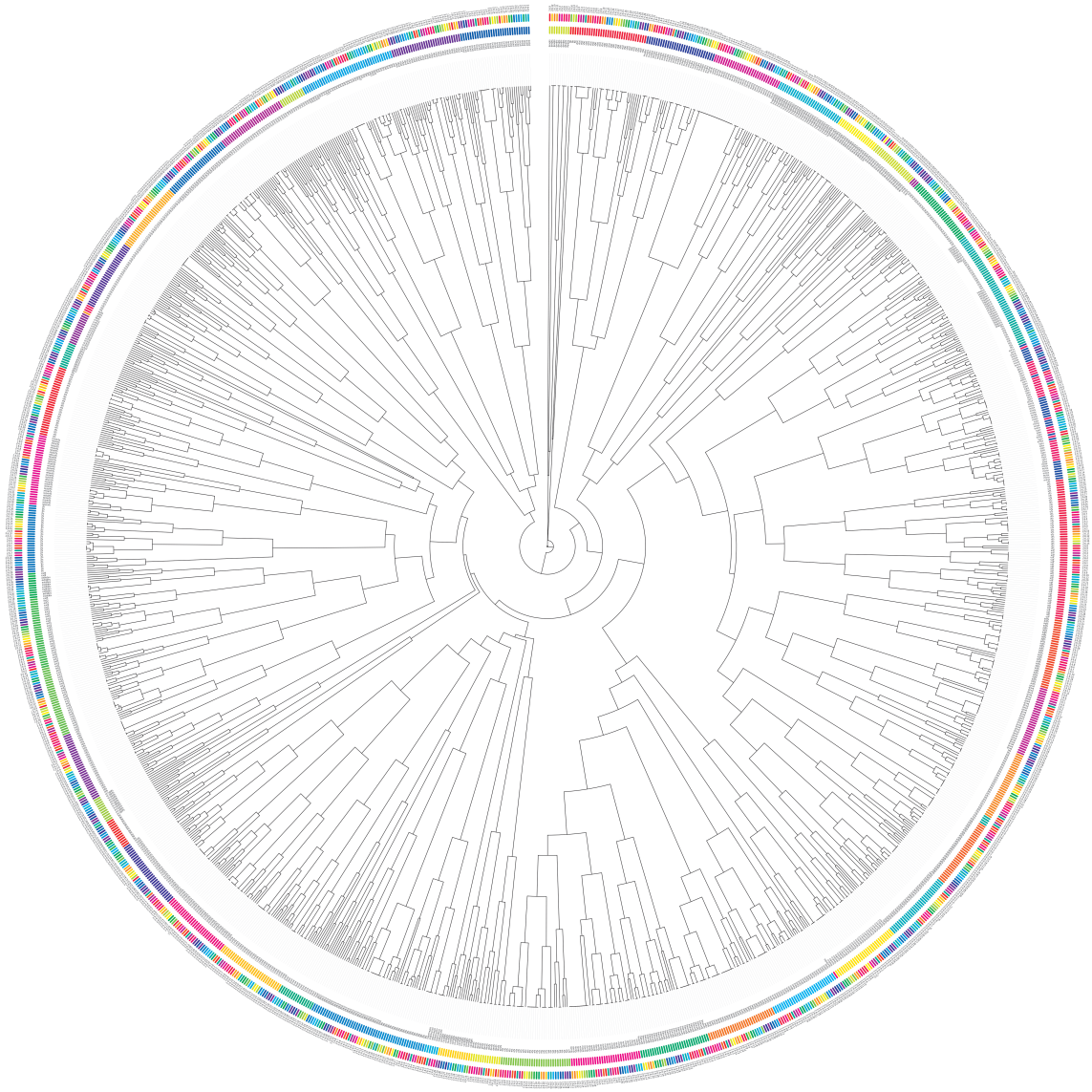
Circular dendrogram of the clustering using Canberra distance and Ward's method. This is the first out of five circular dendrograms that shows the result of clustering using the Canberra distance over all the file systems. As we mentioned before, in our case, this metric is equivalent with the number of different external symbols.



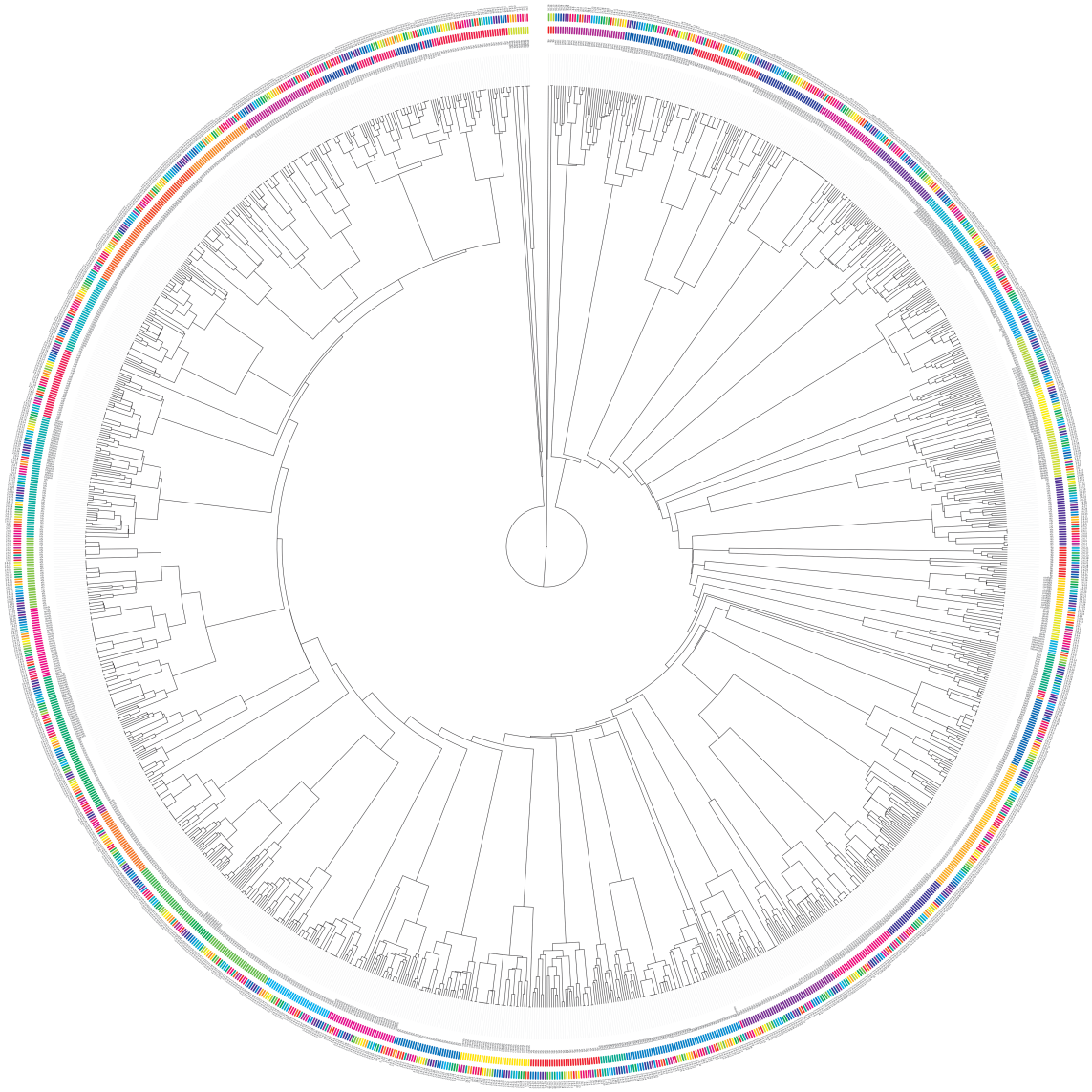
Circular dendrogram of the clustering using Canberra distance and complete linkage.



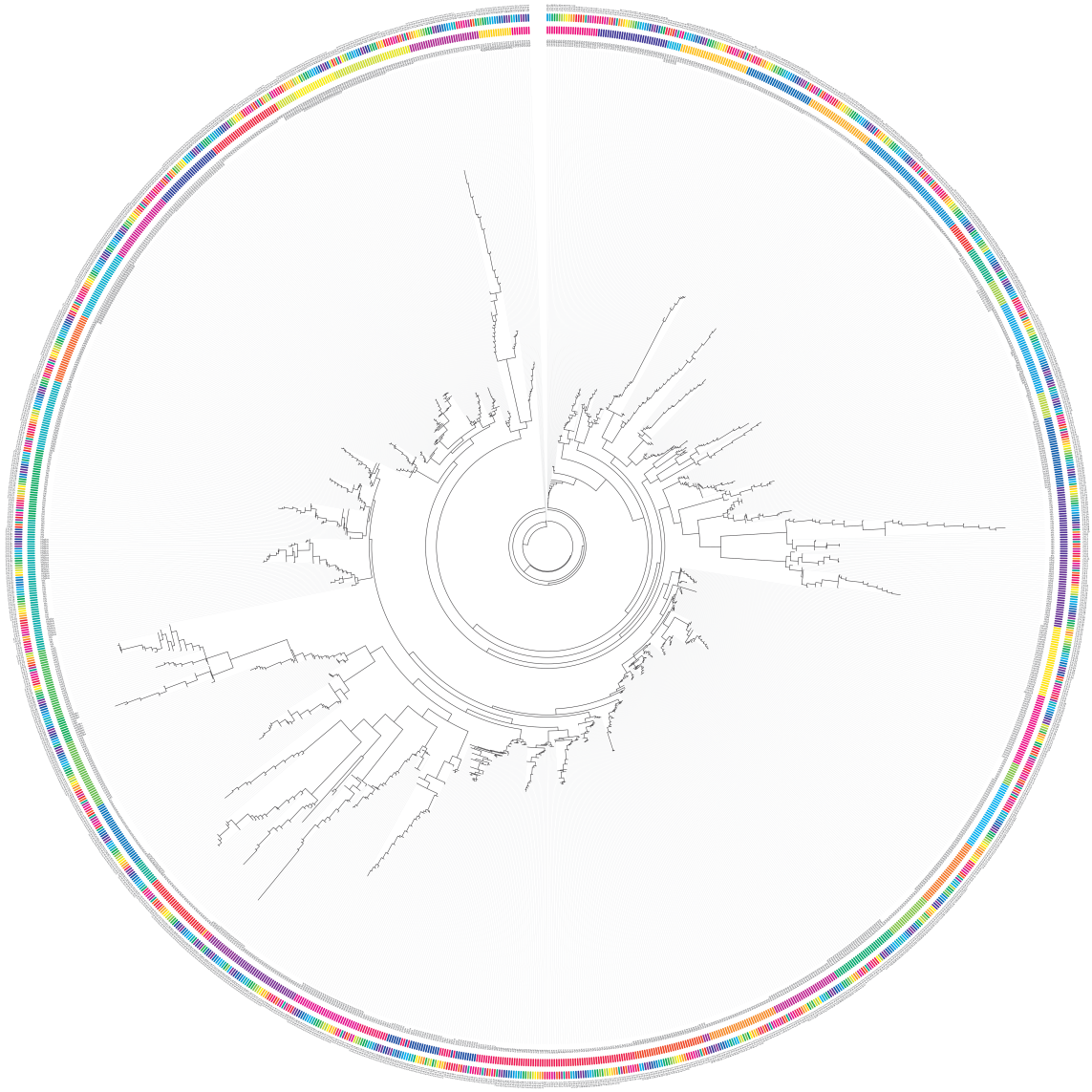
Circular dendrogram of the clustering using Canberra distance and group average.



Circular dendrogram of the clustering using Canberra distance and McQuitty's method.



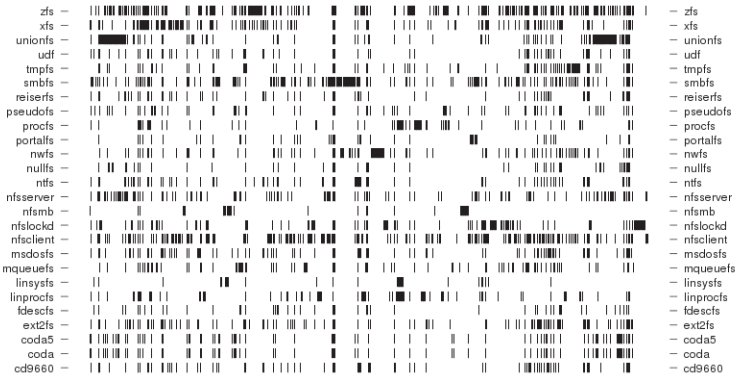
Circular dendrogram of the clustering using Canberra distance and single linkage.



Circular representation of the phylogenetic tree. This last plot shows the phylgenetic tree construct by Pars.

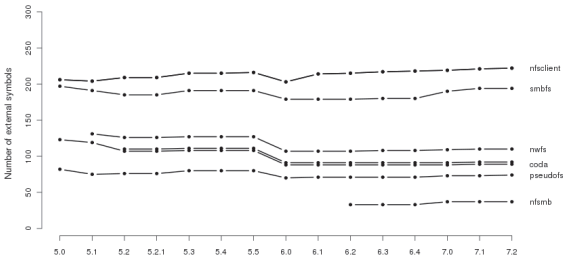
4. The BSD Family

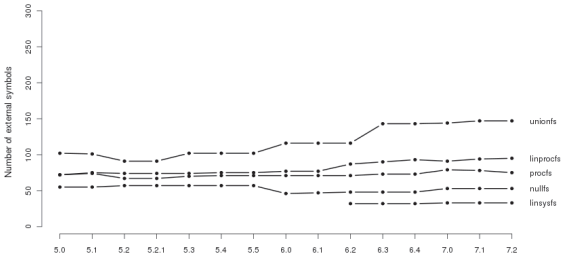
In this section we are going to take a quick look at the BSD world. The operating systems considered are FreeBSD, NetBSD, OpenBSD and Darwin. All the systems have support for kernel modules but their use in NetBSD and OpenBSD is limited. The "over time" term is used here to refer to releases. Timelines for all the systems except Darwin are presented in Appendix B.

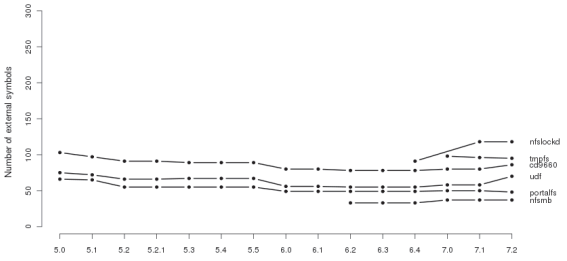


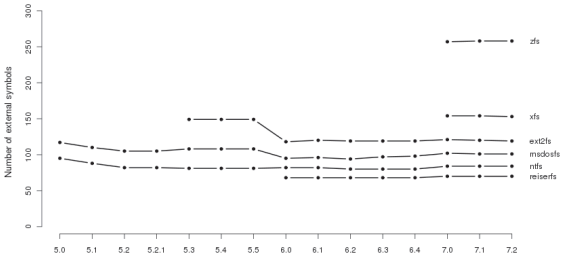
Map of the external symbols for FreeBSD 7.2. As before, each tick represents an external symbol. The horizontal axis contains 832 symbols. The 26 .ko modules related to file systems were taken from the boot/kernel directory from the bootonly flavor of the install CD. The FFS, the native filesystem for FreeBSD is not compiled as module so it doesn't show up here.

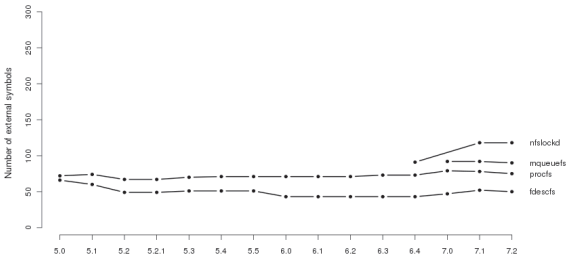
Number of external symbols over time for FreeBSD. To avoid overlapping this plot is split in five parts. A few things are noticeable: first, for most of the modules the number of external symbols dropped from 5.5 to 6.0 and second, with the exception of unionfs all the other file systems didn't change much.



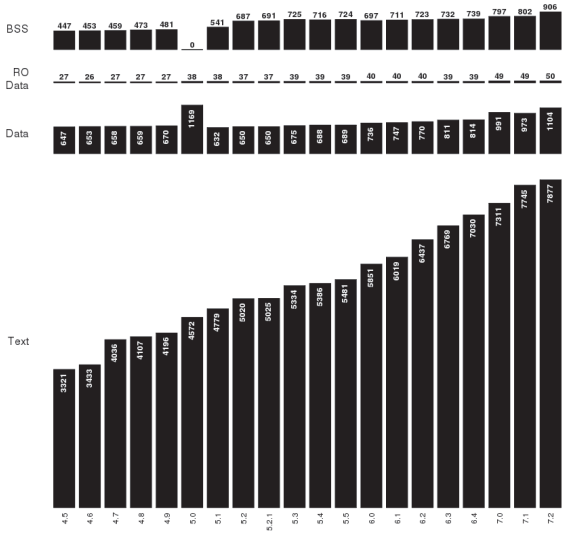




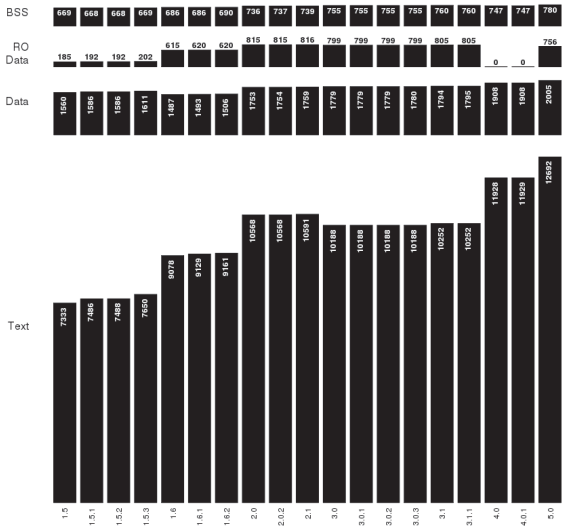




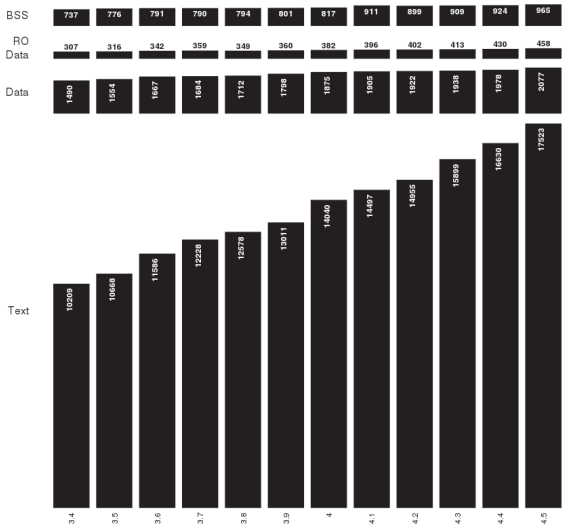
Exported symbols over time for FreeBSD. Because the compiled kernel is readily available, an easy thing to do is to look how the number of exported symbols changed over time. Note that this plot starts with FreeBSD 4.5, the first release for which symbols were not stripped from the kernel.



Exported symbols over time for NetBSD. Unlike FreeBSD, in NetBSD there are significant, usually increase, in the number of symbols from one major release to another. The plot starts from 1.5 because that was the first one being compiled as ELF.



Exported symbols over time for OpenBSD. As in the case of NetBSD, the graph is shorted at the left because all the older ones are a.out.



Map of the external symbols for Darwin 9.7.0. This is the kernel for MacOS X 10.5.7. Each tick represents an external symbols. There are 489 symbols on the horizontal axis.

5. Conclusions

This is the end of this expedition. There are a lot of things that I didn't had time to try. Here are a few of them.

As we saw, the number of external symbols is quite big. Even one file system can have hundreads and we end up with more than a thousand symbols when we consider a decent selection of file systems. This makes tracking each symbol individually not very informative. One way to reduce the complexity is to try to classify them. Doing this manually could be accomplished by somebody familiar with that respective kernel but some automatic method might could be also attempted. With this classification in place, the way the file systems are using various classes of symbols could become more meaningful.

Another direction, which increases complexity, would be to take in consideration not only that a kernel module is using a certain external symbol but also from how many different places from inside its code it is doing this. This information is contained in the relocation table of the object file and it can be easily extracted using objdump.

From the personal side I can say that figuring how to get all the Linux modules was kind of cool and learning about the way hierarchical clustering works was very informative. The fact that BSDs have a system release with each of their kernels (or conversely, that they make a new kernel release with each system release) made them much easier to deal with. Their archives, which contains binaries going all the way back to the very beginning, represents a very valuable resource which could be used to track their evolution.

Some trivia. There are 78 regular figures (out of which 10 have high-detail versions) and 4 animations. The building of the phylogenetic tree for all the 2.6.x took about tree days of continuous running on a P4 at 2.8 GHz. The memory consumption was decent though, only 200 MB. Before settling to the final Circos graph I generated more than 50 circular plots showing the relations between each file systems with everybody else. All of them look very similar though. Except the circos plot and the the treemap all the others are done in R. The treemap was obtained using GrandPerspective and Inkscape.

Thank you for reading! And once again, if you find any mistake please let me know.

Appendix A: The Building Process

Trying to build all the file systems from 2.6.x is quite a challenge in itself due the span of more than 5 years which passed from 2.6.0 (December 2003) to 2.6.29 (March 2009). Luckily, kbuild, the makefile-based build system did not change that much and I had to divide the releases in two groups: 2.6.0 to 2.6.12 (older half) and 1.6.13 o 2.6.29 (newer half). For the first one I had to use the

```
make -C ... SUBDIRS=...module.o
```

command while for the second I was able to use a simpler one

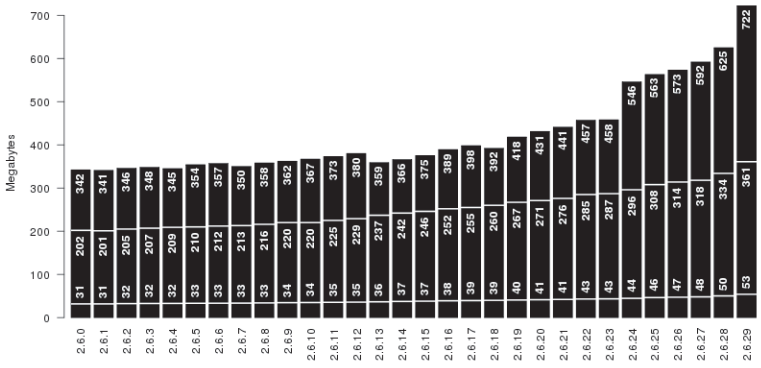
```
make fs/fs/module.ko.
```

In both cases the sequence of configuration commands was the following:

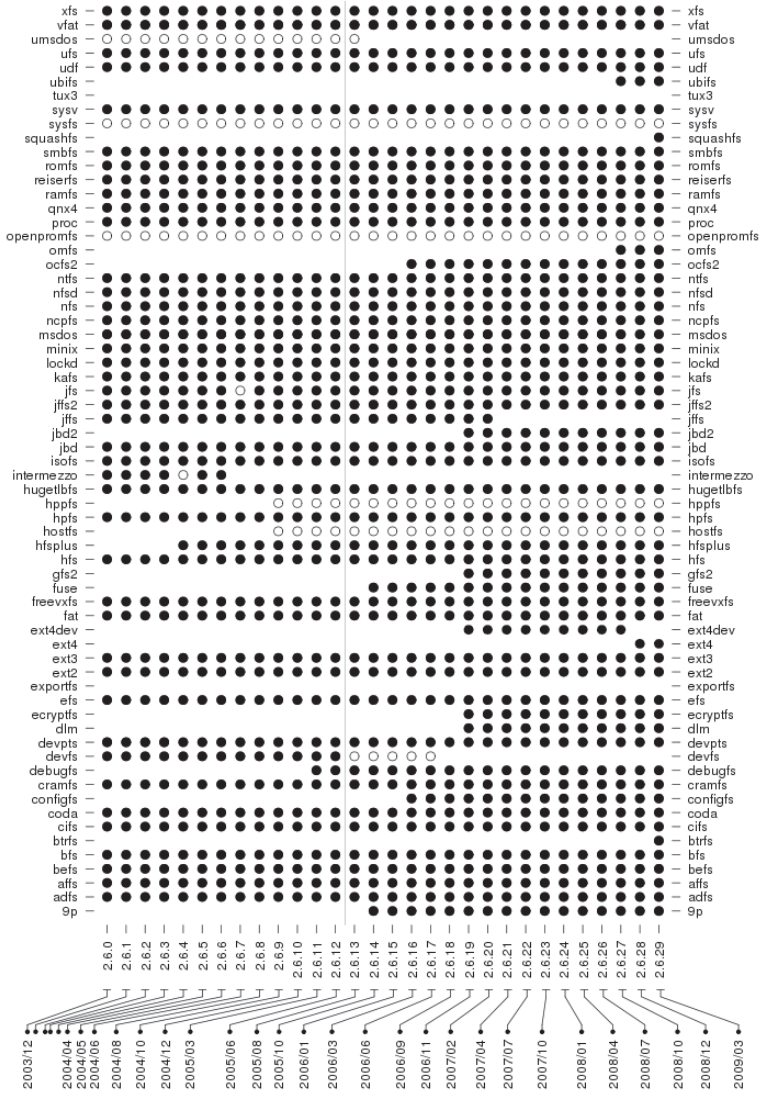
```
make mrproper
make allmodconfig
make prepare
```

On the toolchain front I used the latest binutils (2.19.1) from Debian 5.0 Lenny and two GCC version: the 4.1.2, the latest GCC from the 4.1 series that is available in Debian Unstable Sid for the newer half and a manually-compiled 2.95.3 for the older one. to get 2.6.16 to 2.6.21 to compile I also had to manually add an include for limits.h in the scripts/mod/sumversion.c.

From a storage perspective, the 30 kernel trees weight about 1.1 GB in tar.bz2 form, 7.4 GB unpacked and 12.4 GB when all the file systems are compiled. To put things into perspective, after make bzImage the 2.6.29 is 1.3 GB and gets to 3.8 GB after make modules. This is, of course, an upper bound because the kernel was configured with make allmodconfig. A few more details: the bzImage obtained this way was 3.7 MB and the 2595 .ko files that were produced sum up to a hefty size of 889 MB.

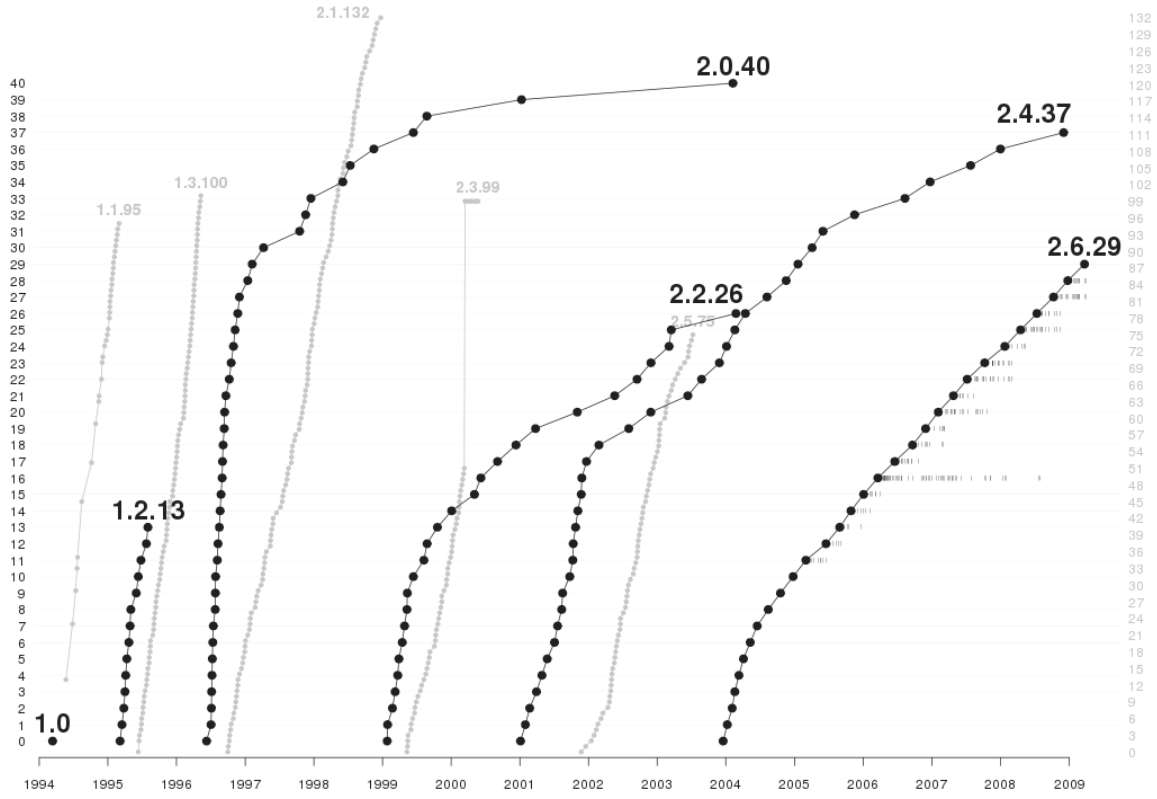


Various sizes of the 2.6.x kernels. From bottom to top, the number indicate: the size of the .tar.bz2 archive, the uncompresses size, the size after all the file systems were compiled.



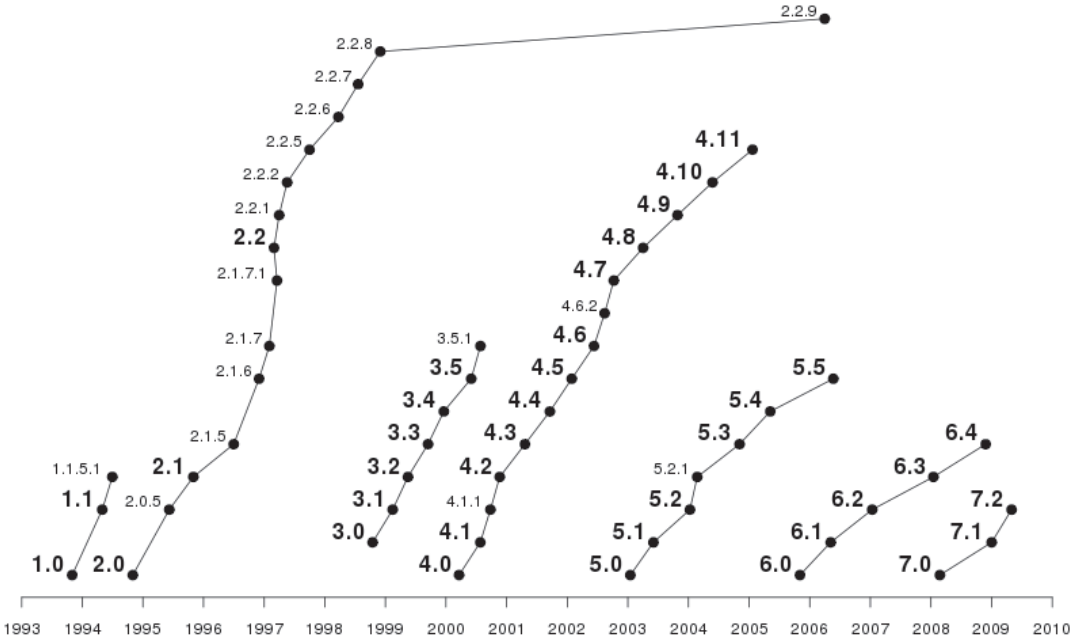
The scorecard of the compilation success. The line between 2.6.12 and 2.6.13 indicates the place where the make command had to be slightly changed. The number of rows is 65. The number of successful compiled file systems for 2.6.29 is 54. The total number of compiled file systems is 1377. ext4 shows up with two names: ext4 and ext4dev. A notable thing is the "no file system left behind" syndrome: once it got in the tree, a file system almost never dies. There is only one true exceptions for this rule: intermezzo. The other two are devfs and jffs which were deprecated in favor of their descendants, sysfs and jffs2.

Appendix B: Timelines



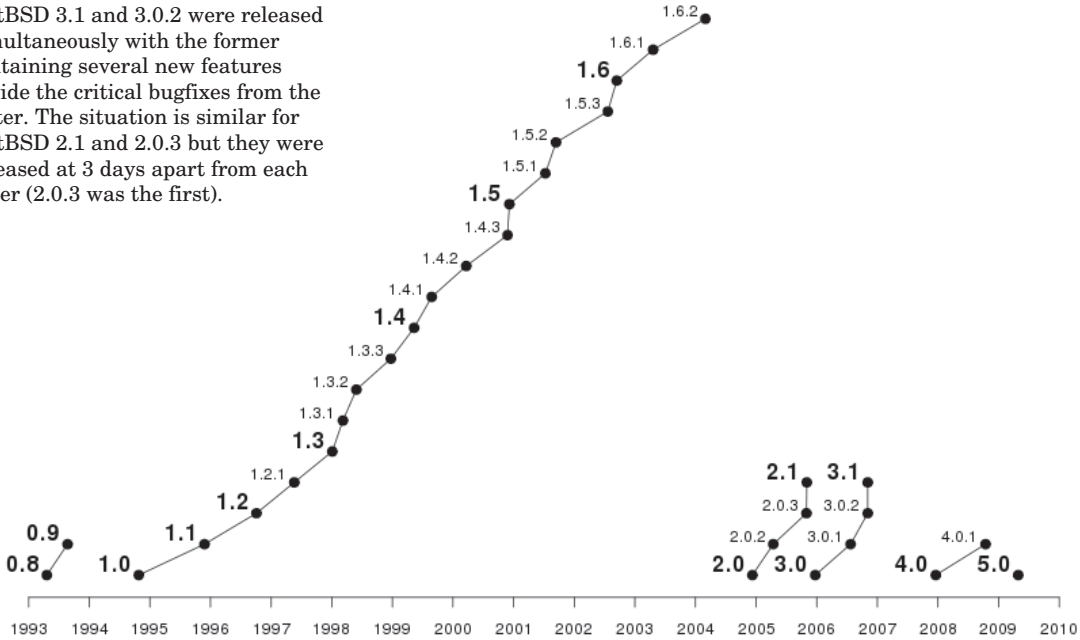
Timeline of the Linux Kernel releases. In gray are depicted the development branches and the small ticks from the the 2.6 branch are the minor releases. The minor that sticks out is the 2.6.16. The data was extracted from the timestamps of the files served by kernel.org. Tho dates, for 2.1.25 and 2.1.26 were wrong and were manually adjusted. The only omitted releases are the 2.2.0pre1 to 2.2.pre9.

Note that the proportion between the left (stable) and right (development) axes is 1 to 3.

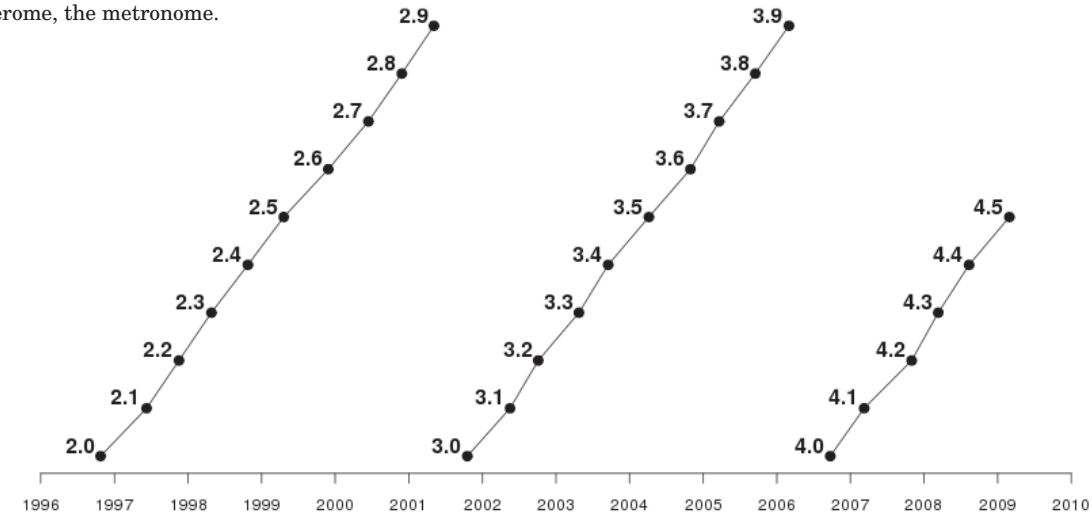


Timeline of the FreeBSD releases. The data for this graph was collected from two sources: the which provides month accuracy dates and also the modification time for the the `x.x-RELEASE/README.TXT` directories. The 2.2.9 release was announce on April 1st, 2006 with a funny message.

Timeline of the NetBSD releases. The NetBSD 3.1 and 3.0.2 were released simultaneously with the former containing several new features beside the critical bugfixes from the latter. The situation is similar for NetBSD 2.1 and 2.0.3 but they were released at 3 days apart from each other (2.0.3 was the first).



Timeline of the OpenBSD releases. The data was collected using the modification timestamps from the x.x/i386/MD5. As the FAQ indicates, the releases are 6 months apart. As Dr. Lamar says in Gattaca: Jerome, Jerome, the metronome.



Appendix C: External symbols for 2.6.29 + tux3

Detailed map of external symbols. This is a complete map of the external symbols used by the file systems compiled from the Linux Kernel 2.6.29 + tux3 branch. The symbols are sorted in the descending order of their frequency while the file systems are sorted in the descending order of number of external symbols they use.

The End.