



File Systems, Unicode, and Normalization

David Robinson, Ienup Sung, Nicolas Williams
Sun Microsystems, Inc.

ABSTRACT

When you try to retrofit existing file systems to support multilingual file names in a predictable manner, you encounter various issues that need to be resolved.

This technical presentation discusses the issues such as why Unicode has to be the choice of the character set for the file systems, how the traditional non-Unicode codesets should be supported, the role of Unicode normalizations, what would be the performance implication, how to deal with possible compatibility and interoperability issues with other file systems and protocols.

Finally, we will also try to provide a comparison on possible resolutions and outcomes as necessary.



Contents

- Traditional File Systems
- Problem Outline
- Goals
- Possible Solutions
- Comparison
- Existing File Systems Using Unicode
- Preferred Solution
- Concluding Remarks
- Q&A



Traditional File Systems

- Usually, traditional file systems allow any octet values at file names as long as they are “file system safe.”
 - > In Unix file systems, valid file names are any that adhere to the C programming language convention that the *NULL byte is the string terminator* and also *do not include path separator*.
 - > Examples:
 - “test.txt”
 - “αβψάβçφπς 한글漢字にはほんご.txt”
 - “\x01\x02\x03\x04-\x80\x81.GF1”
- Generally, there are no per-file or per-file system attributes or tags identifying codesets.

Page 3 of 27

As you can see in the Examples shown at the slide, traditional file systems usually allow any octet or byte values in file names as long as they are file system safe. This level of flexibility and freedom has been achieved by making the file system and related kernel modules and system libraries 8-bit clean.

In the traditional Unix file systems, the “file system safe” means that the file names do not include the path separator character (‘/’ or 0x2F in 7-bit ASCII character code) and the NULL character (‘\0’ or 0x0) is used only as the file name string terminator.

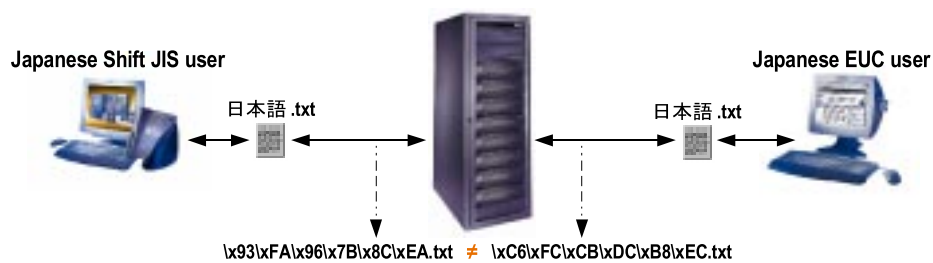
There are also usually no per-file or per-file system attributes that can be used to tell what is the codeset being used with a file name or a file system.



Problem Outline

8-bit clean alone ≠ internationalized

- Users using different codesets cannot interoperate either between or within a file system:
 - > No codeset tags on file names.
 - > Applications and users must “guess” correct interpretation.



Page 4 of 27

The simplicity (and freedom) described in the previous slide allows people to create and use almost any kind of file names with their applications and, at the same time, it also introduces a problem of difficult interoperability or interchangeability of file names among different people or processes.

As an example, in a Japanese company where there are several different kinds of codesets (such as Shift JIS, Japanese EUC, UTF-8, and so on) co-exist with heterogeneous user environments, people have difficulty sharing their files if Japanese characters are used in the file names since even though they are the same Japanese characters, code point values are different with different codesets.

In such environments, applications and users have to decide on how to interpret the file names and their codesets. This “decision” has to be continuously re-stated and complied with since there are always new applications and new users.

This level of limitation and difficulty exists not only in Asian countries but also in any other regions of the world. For instances, in Western Europe, there are ISO 8859-1, ISO 8859-15, UTF-8, Windows-1252, and so on; in European Union, there are many single byte and multibyte codesets being used.



Goals

- Enhance traditional file system components to *transparently* provide *multilingual-capable* and *predictable* file name codesets:
 - > Support interoperable multilingual file names.
 - > Codeset details transparently abstracted from users and applications.
 - > Stored in locale and codeset independent format.
 - > Preserve binary compatibility.

Page 5 of 27

Our goals at the moment are shown in this slide.

We hope to achieve the goals with the maximum transparency possible and also in binary compatible way so that there will be no need to re-compile existing binaries and yet people will be able to utilize the feature with their existing applications.



Possible Solutions

- What is the codeset of a file name?
 - > Per-file codeset attribute.
 - > Unicode as the default codeset.

Page 6 of 27

To achieve the goals, naturally, we must know the codeset of each file name. We believe there are two major approaches possible for that:

1. Add per-file attributes that will tell codeset (and optionally also language and territory) of each file name, or,
2. Use Unicode as the default and canonical codeset of file systems for all file names.



Possible Solutions

Adding per-file attributes

- File system module or upper layer software uses attributes to perform necessary code conversions for proper representation.
- Requires on disk directory entry data structure changes breaking backward compatibility.
- Significant code changes.

Page 7 of 27

With the additional data attributes on file names such as codeset, language, and territory values, file system modules or upper layer software could do necessary code conversions for proper representation of the file names.

This approach, however, may require changes in existing data structures on directory entry that is on disk and also compiled into existing application binaries causing a breakage on binary backward compatibility.

As an example, Unix systems usually declare and use something similar like the following file system independent directory entry data structure:

```
typedef struct dirent {
    ino_t      d_ino;          /* "inode number" of entry */
    off_t     d_off;          /* offset of disk directory entry */
    unsigned short d_reclen;  /* length of this record */
    char      d_name[1];     /* name of file */
} dirent_t;
```

Adding a new data field or augment something new into the above like fixed data structure could cause binary backward compatibility breakage and possibly also could require a new set of data structures and functions.

It will also cause significant amount of code changes at file system modules and upper layer software including not only system libraries but also possibly user applications.



Possible Solutions

Using Unicode as the default

- Simplifies code conversions from many-to-many to many-to-one.
- Maintains existing on disk 8-bit clean data structures.

Page 8 of 27

On the other hand, by assuming that all file names are in Unicode (especially UTF-8), file system modules or upper layer system software can do necessary code conversions transparently for the proper representation of file names. The number of necessary code conversions will also be reduced significantly.

Since there are no additional data fields for additional attributes, there is no binary backward compatibility issue on the file system data structures.

Unicode is the best universal character set that is widely accepted and implemented by numerous software and many modern standards. It also comes with various normative and informative information that are useful. We believe choosing or inventing anything else is simply not necessary. Furthermore, Unix file systems already support UTF-8.



Possible Solutions

Comparison

	Adding per-file attributes	Using Unicode as the default
Binary backward compatibility	No	Yes
Complexity of implementation	Relatively complex	Simple
Performance	Slow	Fast
The original file names?	Yes	Possibly No
Could have additional attributes?	Yes	No

Page 9 of 27

By using Unicode as the default and canonical codeset, we believe we can design and implement the feature that will fulfil the goals.

As a drawback, however, we may not be able to retrieve the original file names unless the codesets used are either one of the Unicode encoding forms or 7-bit ASCII since, otherwise, there will be code conversions and the outcomes of the code conversions may yield equivalent but not necessarily the same file names.

Since there are no new data structures or changes, the approach will not be able to accommodate any new data attributes such as language or territory that could be useful in certain extent.



File Systems Using Unicode

Microsoft: NTFS and VFAT/LFN

- NTFS and VFAT/LFN use Unicode as the file name codeset:
 - > UTF-16/UCS-2.
 - > No Unicode Normalization or any other special treatments on the file names.
 - > NTFS long file names can be up to 255 16-bit units.
 - > Since long file names can be up to 255 16-bit units, a LFN could take up to 20 LFN directory table entries since each LFN entry can have thirteen 16-bit units in maximum.

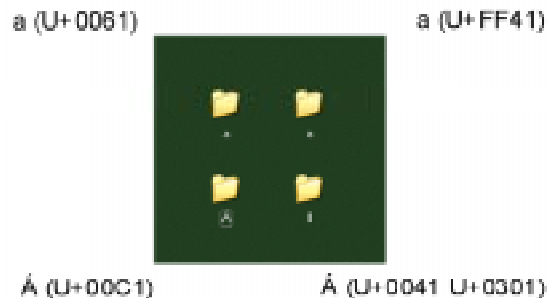
Page 10 of 27

We consider interoperability with other file systems quite important.

In this and following a few slides, we hope to show some notable characteristics and differences of existing file systems from others. Interestingly, all recent file systems use Unicode as the default codeset of the file systems.

Windows NT file system (NTFS) and Long File Name (LFN) used in Virtual File Allocation Table (VFAT) file system use Unicode (UTF-16/UCS-2) as the file name codeset.

After some experiments with a Windows XP system, we think all Unicode file names are created on storage mediums without any special treatments or Unicode Normalizations as long as the characters are valid Unicode characters. The following screen snapshot shows there are four file names created on the "Desktop" that would not be possible if Unicode Normalizations had been performed before the file name creations:



The LFN implementation is an interesting way of retrofitting existing file systems but we believe that is not an approach that we want to adopt into our file systems.



File Systems Using Unicode

Apple: HFS+

- Apple MacOS 8.1 and later uses Unicode for their file and folder names with HFS+:
 - > UTF-16 (up to 255 16-bit units).
 - > Uses Unicode Normalization Form D (NFD).
 - > Case-insensitive comparison on the file and the folder names.
 - HFSX, a variant of HFS+, can be enabled to be case-sensitive.
 - > Has “textEncoding” data field in each catalog file/folder record.
 - A “hint” during the conversion from Unicode to MacOS-encoded Pascal string.

Page 11 of 27

Hierarchical File System Plus (HFS Plus or HFS+) is another Unicode file system that is being used as the primary file system at MacOS.

It uses Unicode Normalization Form D (NFD, canonically decomposed) and does case-insensitive file name comparisons.



File Systems Using Unicode

OSTA: UDF

- UDF (Universal Disk Format) uses OSTA CS0, i.e., d-characters of the Unicode 2.0 except U+FEFF and U+FFFE:
 - > Maximum of 255 bytes in “OSTA Compressed Unicode” format, i.e., up to 127 or 254 characters.
 - > No Unicode Normalization or any other special treatments on the file names.

Page 12 of 27

Optical Storage Technology Association (OSTA) defines Universal Disk Format (UDF) that is widely used in DVDs and some CDs.

The UDF Rev. 2.60 shows that OSTA CS0 character set shall consist of the d-characters of the Unicode 2.0 except U+FEFF and U+FFFE (i.e., no byte ordering mark) and stored in the “OSTA Compressed Unicode” format shown in the following table:

Relative Byte Position	Length	Name	Contents
0	1	Compression ID	UInt8
1	??	Compressed Bit Stream	Byte

The Compression ID identifies the compression algorithm used in the Compressed Byte Stream and currently there are basically two compression algorithms: Compression Algorithm 8 which just takes lower 8 bits of Unicode wide character values and Compression Algorithm 16 which takes lower 16 bits of Unicode wide character values. Based on what kind of compression algorithms are used, a file name can have up to 127 or 254 Unicode 2.0 d-characters.

It appears there are no other Unicode Normalization or special treatments defined in the UDF Rev. 2.60 specification.



File Systems Using Unicode

Linux: FAT/VFAT, NTFS, UDF, and the rest

- In kernel per file system code conversions for VFAT/LFN, NTFS, and UDF.
 - > Mount options like “utf8,” “codepage,” “iocharset,” and “nls” are needed for the correct conversions.
- For any other file systems, it is up to each application and user.

Page 13 of 27

Some of the Linux file system implementations have mount options that can be used to tell what kind of input/output charset, codepage, or whether it should interact with UTF-8.

With that information, some pre-defined code conversions can be supported transparently. For any other file systems such as ext?, hfs, reiserfs, ufs, and so on, it appears that it is up to each application and user on how to interpret file names.



Comparison

File System	Unicode Encoding	Maximum File Name Length*	Unicode Normalization?	Case Sensitive?
NTFS	UTF-16	up to 255	No	Yes
VFAT/LFN	UCS-2†	up to 255	No	Yes
HFS+	UTF-16	up to 255	Yes (NFD)	No‡
UDF	Unicode 2.0 [!]	up to 127 or 254 ^{!!}	No	Yes

* In 16-bit unit unless noted otherwise.

† It might be UTF-16 but we are not sure.

‡ With HFSX, it can be enabled to be case-sensitive.

! It can be in either 8-bit or 16-bit encoding.

!! 127 when using Compression Algorithm 16; 254 when using Compression Algorithm 8.

Page 14 of 27

As you can see from the table, there are some notable differences among the Unicode file systems:

1. HFS+ does NFD while others do not.
2. Unlike others, HFS+ does case-insensitive file name comparisons.

Even though they are shown in the comparison table, the differences on the used Unicode encoding forms or versions and also the maximum file name lengths do not have any significant consequences or implications in our design and implementation. The important things are that they are all supporting Unicode with reasonable maximum file name length values.



Preferred Solution

- Use Unicode as the default and canonical codeset of Solaris file systems:
 - > Encoding form will be UTF-8.
 - > New “utf8” mount option enables Unicode.
 - > C library (libc) will do transparent code conversions on file names when applicable (configurable).
 - Will use PUA/UDC/VDC mappings supplied by the iconv code conversion subsystem.
 - > File names will not be normalized but no two file names in a directory will be allowed to be canonically equivalent.
- File system migration tools will be supplied.

Page 15 of 27

We believe using Unicode as the default and canonical codeset of Solaris file systems is a natural thing to do. A few key characteristics of the feature are:

1. Encoding form will be UTF-8.
2. Users can enable the Unicode/UTF-8 file system feature by supplying “utf8” mount option.
3. The libc will do transparent code conversions on file names so that code conversion details are hidden. This transparent code conversion will be configurable so that when necessary, it can be turned off, for instance. We will also use PUA/UDC/VDC mappings supplied by iconv code conversions. (The iconv code conversions are user customizable.)
4. We will not store normalized file names. However, we will use Unicode Normalization NFD to check on canonical equivalence among file names in a directory so that there will be no two file names in a directory that are canonically equivalent.

To aid users who want to migrate their file systems, we will also provide file system migration tools.



Preferred Solution

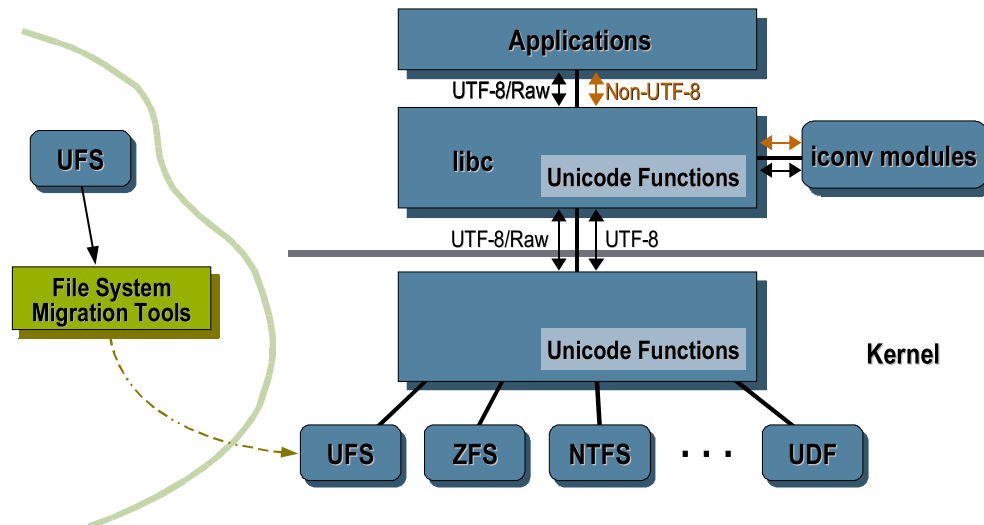
- Existing Unicode file systems will be treated with the maximum compatibility possible. For example:
 - > HFS+ file names will be NFD normalized.
 - > NTFS and non-NTFS file names will be compared byte by byte.

Page 16 of 27

We will also ensure the maximum possible compatibility between our file systems and other Unicode file systems.



Preferred Solution



Page 17 of 27

As you can see from the architecture diagram in this slide, the code conversions and other Unicode related functionalities like Unicode Normalizations and canonical equivalence checks are hidden from users and applications.

The transparent code conversions will be done only when the current codeset of the locale is not 7-bit ASCII or UTF-8 which is noted with orange colored arrows with "Non-UTF-8" in the diagram. All other Unicode related processings will also be performed only when they are necessary. If the current locale's codeset is UTF-8 (including 7-bit ASCII) or use of raw data (i.e., any arbitrary character code point values) at file names are desired, they can pass through to the file system modules without any code conversions, Unicode specific treatments, or both.

When dealing with other Unicode file systems, that particular Unicode file system specific rules will be observed and complied for the maximum compatibility and interoperability.

Existing Solaris file systems can be turned into Unicode/UTF-8 enabled file systems by utilizing the file system migration tools.



Preferred Solution

Why UTF-8 as the default codeset of file systems?

- UTF-8 is the natural choice since it has been developed as a Unix file system safe Unicode encoding form from the start.
- By assuming UTF-8, there will be no binary backward compatibility issues on directory and file system data structures.
- More economical to design and implement than any other options.

Page 18 of 27

As briefly mentioned in the slides 8 and 15, UTF-8 is the natural choice since it is Unix file system safe and has been supported and used with our file systems for many years. (The original name of UTF-8 was UTF-FSS which stands for Universal Transformation Format-File System Safe.)

Since it is currently supported and also requires no changes at any of existing data structures, there are no binary backward compatibility issues. It will be much more economical to design and implement than any other options.



Preferred Solution

Why the “utf8” mounting option?

- To keep the backward compatibility and provide options to users so that they can choose to migrate when most feasible.

Page 19 of 27

While the feature will be binary backward compatible, there are certain things such as additional internal processings described in the slides 15 and 17 that could be seen by users as something that are not necessary for their applications and also do not want to have. We are also anticipating that due to almost inherent intransitivity nature of some code conversions, there will be some rare cases where the original context and meaning of file names could be lost in the code conversions.

And thus even though we consider such “backward compatibility issues” are minor, the ultimate decision should be made by the users and, for that, we are introducing the “utf8” mount option.

When it is present, the UTF-8 file system feature will be enabled; if not, the feature will not be enabled and there will be no differences on the file systems from the user's perspective between older Solaris releases and the next major Solaris release.



Preferred Solution

Transparent code conversions at libc

- The code conversion will be done at file name related I/O functions and also system calls between the current locale's codeset and UTF-8 *iff* the file system is a UTF-8 file system and user didn't "turn off" the feature.
 - > If the current locale's codeset is UTF-8 or 7-bit ASCII, there will be no code conversions.
 - > Feature may be controlled by an environment variable.
- Invalid UTF-8 or raw data file names will be allowed unless explicitly prohibited by open or create functions.

Page 20 of 27

Code conversion between current locale and UTF-8 will be done at the system call stub layer at libc. Unicode encoding conversions between UTF-8 and file system specific encoding forms will happen at the Virtual File System (VFS) boundary when necessary.

By using an environment variable, the transparent code conversion can be controlled.

We will also allow invalid UTF-8 or raw data file names unless such are explicitly prohibited by users. (Users who want to prohibit such names can supply a new flag value to file open or create functions.)



Preferred Solution

File name normalizations

- File names will be saved without any processing such as Unicode Normalization.
- However, file names will always be compared with canonical equivalence check so that no two names that are canonically equivalent can co-exist in a directory.
- This will ensure that there will be no confusion over the canonically equivalent file names (that will look identical) in the same directory.

Page 21 of 27

As noted earlier, in Solaris file systems, no file names will be normalized. However, to reduce any possible confusions on canonically equivalent file names, we will always make sure to run canonical equivalence checks before we create a file name so that there will be no two file names that are canonically equivalent at a directory under normal circumstances.

The canonical equivalence check will be based on the Unicode Normalization NFD.



Preferred Solution

File system migration/conversion tools

- Tools can be used to update file systems so that existing file systems can be turned into UTF-8 file systems.
- They will be able to do many things automatically but human interactions will be required in many cases when it cannot determine about the codeset of existing file names.

Page 22 of 27

Solaris already has GUI based file system migration tool, `fsexam(1)`. We hope to enhance further on this tool and also possibly supply another file system migration tool that can be used to update and convert existing file systems into UTF-8 file systems.

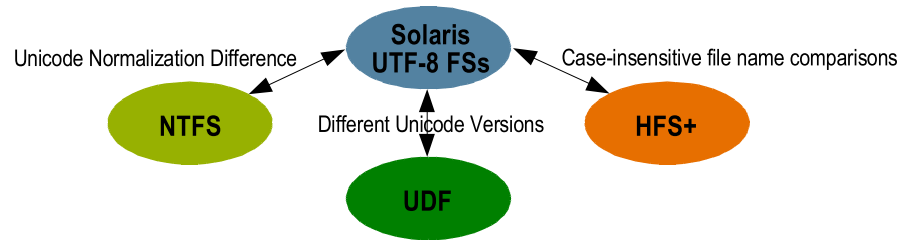
Even though these tools are designed to auto-detect and auto-convert file names into UTF-8 as much as possible, there will be cases where these tools will not be able to do the job completely and correctly by themselves. For such cases, users will have to manually supply decision makings and possible rollbacks if necessary.



Preferred Solution

Compatibility & interoperability with other Unicode file systems

- The major compatibility and interoperability issues identified are shown at the following diagram:



Page 23 of 27

We consider compatibility and interoperability between Solaris file systems and other Unicode file systems very important. The identified issues are shown in this slide.



Preferred Solution

Compatibility & interoperability with other Unicode file systems cont'd

- NTFS:
 - > When comparing with NTFS file names, no canonical equivalence check but byte-by-byte comparison will be used.
- HFS+:
 - > When comparing with HFS+ file names, case-insensitive comparison will be used.
 - > Before a file name is created, it will be normalized with the NFD.
- UDF:
 - > Unicode version differences will be ignored considering that the most frequently used characters are still in the BMP area.

Page 24 of 27

Especially, we hope to make sure the above mentioned compatibility implementations are in places and our file system modules will observe the rules and requirements defined by other vendors and standard bodies as much as possible.



Preferred Solution

Performance implication

- Considering the code conversions and the Unicode Normalizations can be done in $O(\log n)$ or $O(n)$ where n is the number of characters in a file name, the performance degradation will be quite negligible.

Page 25 of 27

Since most of the already existing Unicode related code conversions and also the Unicode Normalization NFD are implemented in $O(\log n)$ or $O(n)$ pretty much, we believe there is no significant performance degradation if users choose to enable the UTF-8 file system feature.



Concluding Remarks

- Unicode file systems with UTF-8 as the default and canonical codeset is the ideal and economic way to retrofit existing Unix file systems.
- Providing transparent code conversions and a way to opt out of this new feature is important; providing binary backward compatibility is also a must.
- A canonical equivalence check is available to ensure less confusions on the file names.
- Complete compatibility and seamless interoperability with all other existing Unicode file systems appears not 100% possible due to inherent differences.



Q&A

Page 27 of 27