GTK+/GLib のファイル名エンコーディング

岩本一樹
iwm@maid.org

概要

GTK+および GLib が扱う文字列は UTF-8 であるが、GTK+と GLib の一部の API は UTF-8 以外の文字列を扱う。また一部の API は暗に UTF-8 を他の文字符号化方式に変換する。本論文では GTK+と GLib の API の文字列の文字符号化方式(Character Encoding Scheme)を検証し、file name encoding を中心に GTK+および GLib でプログラムを組む上で注意すべき点をまとめる。

1.ファイルやパスを表す文字列

ほとんどの API では文字列は UTF-8 であるが、ファイルやパスを表す文字列は UTF-8 以外の場合がある。GLib Reference Manual の Character Set Conversion ではそれを file name encoding と言っている。UTF-8 と file name encoding の関係は GLib Reference Manual の Character Set Conversion で説明されている。open や stat、readdir のような元々 GTK+や GLib と関係ない API が UTF-8 ではないのは言うまでもないことである。しかし GTK+や GLib の API であっても、主に前述の open や stat、readdir のような API に使われるファイルやパスを扱う GTK+や GLib の API は UTF-8 以外の時がある。

例えば GtkFileChooser で選択されたファイル名は gtk_file_chooser_get_filename で取得できるが、この API が返す文字列は file name encoding である。また GtkFileChooser にファイル名を設定する gtk_file_chooser_set_filename の引数の文字列は file name encoding である。

実際の file name encoding の文字符号化方式は GLib-2.4 以降は環境変数 G_FILENAME_ENCODING で指定される。GLib-2.6 以降ならば環境変数 G_FILENAME_ENCODING は複数の文字符号化方式を「,」で区切って指定できる(GLib-2.4 であっても「,」で区切る記述は認められるが、先頭の文字符号化方式のみ有効となる)。

例
G_FILENAME_ENCODING=ISO-8859-1

例
G_FILENAME_ENCODING=EUC-JP,CP932

文字符号化方式に@locale を指定した時にはそれはロケールの文字符号化方式になる。ロケールの文字符号化方式とは、優先順位が高い順に nl_langinfo の返値、setlocale の返値、環境変数の LC_ALL、LC_CTYPE、LANG の値で示されるいずれか1つの文字符号化方式のことである。Microsoft Windows では GetACP の返値が元になる。

例
G_FILENAME_ENCODING=@locale,CP932

もし環境変数 G_FILENAME_ENCODING が定義されていないか、GLib-2.4 より前で環境変数 G_BROKEN_FILENAMES が定義されているならば、file name encoding の実体は g_get_charset の返値になる。すなわち現在のロケールの文字符号化方式になる。

環境変数 G_FILENAME_ENCODING と G_BROKEN_FILENAMES の両方が定義されていない時には、file name encoding の実体は UTF-8 になる。

バージョン	G_FILENAME_ENCODING	G_BROKEN_FILENAMES	文字符号化方式
2.2 以前	無視される	あり	ロケールの文字符号化方式
		なし	UTF-8
2.4	あり	無視される	G_FILENAME_ENCODING の 文字符号化方式(単一)
	なし	あり	ロケールの文字符号化方式
		なし	UTF-8
2.6 以降	あり	無視される	G_FILENAME_ENCODING の 文字符号化方式(複数可)
	なし	あり	ロケールの文字符号化方式
		なし	UTF-8

なお Microsoft Windows ではどのバージョンも環境変数 G_FILENAME_ENCODING と G_BROKEN_FILENAMES には対応しておらず、これらは無視される。Microsoft Windows における file name encoding の実装は後に述べる。

2.file name encoding の変換

これらの変換のために GLib の API には g_filename_to_utf8 と g_filename_from_utf8、GLib-2.6 以降ならば g_filename_display_name、g_filename_display_basename がある。g_filename_display_basename は引数のファイル名のディレクトリを取り除いた名前部分だけを UTF-8 に変換して返す。

g_filename_to_utf8 と g_filename_display_name の違いは変換を試みる文字符号化方式にある。GLib-2.6 以降には g_get_filename_charsets という API があり、これは file name encoding の実際の文字符号化方式を返す。もし環境変数 G_FILENAME_ENCODING で複数の文字符号化方式を指定していたならば、この API は複数の文字符号化方式を返す。

g_filename_to_utf8 は g_get_filename_charsets が複数の文字符号化方式を返しても最初の文字符号化方式からのみ UTF-8 への変換を試みる。それに対して g_filename_display_name は複数の文字符号化方式が返された時には順番に変換が成功するまで変換を試みる。ゆえに g_filename_display_name の方が UTF-8 に変換できる可能性が高いと言える。しかし g_filename_display_name で UTF-8 に変換されたファイル名は、どの文字符号化方式で変換されたのかわからないため file name encoding に戻す変換はできない。g_filename_to_utf8 で変換されたファイル名は g_filename_from_utf8 で UTF-8 から file name encoding への変換ができる。しかし g_filename_to_utf8 と g_filename_from_utf8 を使った変換では元の同じ文字に変換できるという保証はない。4 節で詳しく述べる。

3.ファイル名はプログラム内部で file name encoding のまま保持する

1 つのプログラムで UTF-8 と file name encoding の 2 種類の文字符号化方式が混在するのは面倒なので、プログラマは gtk_file_chooser_get_filename や g_dir_read_name で文字列を取得したならば、直ちに UTF-8 に変換して file name encoding の文字列をその直後に破棄したくなるかもしれない。file name encoding が必要ならば、そのときに UTF-8 から file name encoding に変換して、不要になったらすぐに file name encoding の文字列を破棄すればプログラムの中のすべての文字列は UTF-8 になる。このアイデアはシンプルで良いように見えるが、問題を含んでいる。file name

encodingとUTF-8の変換が失敗した時、プログラムが正常に動作しなくなる。
　file name encodingをg_filename_display_nameやg_filename_to_utf8でUTF-8に変換できなかったとしても、そのファイル名が間違っているというわけではない。変換は失敗するかもしれないが、だからといってgtk_file_chooser_get_filenameやg_dir_read_nameが返した文字列をopenやstatのようなAPIで使えないわけではない。プログラム内部ではfile name encodingのファイル名を保持し続け、openやstatのようなAPIを呼び出す時には元々のfile name encodingのファイル名を使うようにする。もし変換ができなかった時には、例えばファイル名の代わりに「Unknown file name」のような文字列を表示する。file name encodingとUTF-8の変換に失敗する原因は

1.環境変数G_FILENAME_ENCODINGとG_BROKEN_FILENAMESが正しくない
2.ファイル名の文字符号化方式がUTF-8との変換に対応していない
3.対応する文字がない

が考えられる。

4.libiconvのマッピング問題

　g_filename_to_utf8でfile name encodingの文字列をUTF-8に変換してからUTF-8の文字列をg_filename_from_utf8でfile name encodingの文字列に戻した時に、最初の文字列と同じ文字列に戻ることは保証されない。g_filename_to_utf8とg_filename_from_utf8の動作はiconvに依存する。例えばiconvではCP932→UTF-8→CP932で変換を行った場合に元の文字に戻らない場合がある。

例
≒:87 90 → E2 89 92 → 81 E0
僴:ED 61 → E5 83 98 → FA 7D
兗:ED 62 → E5 85 8A → FA 7E
黑:EE EC → E9 BB 91 → FC 4B
∵:FA 5B → E2 88 B5 → 81 E6

5.設定ファイルに保存

　プログラムを利用するユーザが開いたファイルや入力した文字列の履歴をファイルに保存し、次に起動した時にもその履歴が生かせるようにするのは珍しいことではない。それはGLibのKey-value file parserを使って実装されるかもしれないし、それ以外の方法で実装されるかもしれない。いずれにしても、設定をテキストファイルに保存するならばUTF-8とfile name encodingの問題が生じる。
　ユーザが入力する文字はUTF-8であるが、ファイル名はfile name encodingである。これらを同じファイルに書き出せば、1つの設定ファイルに異なる文字符号化方式が混在することになる。設定ファイルをエディタで開く場合、UTF-8のテキストファイルとして読み込めばファイル名は文字化けするし、file name encodingのテキストファイルとして読み込めばファイル名以外の文字は文字化けする。
　解決方法の1つとしてUTF-8またはfile name encodingのどちらかの文字列をg_strescapeにより7ビットの範囲に収めてしまう方法が考えられる。この場合はテキストエディタで矛盾なく編集できるが、可読性が低下する。
　もう1つはfile name encodingの文字列をUTF-8に変換して保存する方法である。この場合には設定ファイルをUTF-8のテキストファイルとして開くことができるが、前述の問題が発生する。

設定ファイルにファイル名を保存する場合には、下記の 3 つから選択することになる。

1.文字列を g_strescape/g_strcompress により変換する
2.file name encoding を UTF-8 に変換する
3.テキストファイルとしての編集を諦める

6.Microsoft Windows のファイル名実装

Microsoft Windows の API は文字列を ANSI コードページで扱うものと、UNICODE で扱うものの両方が存在する。ANSI コードページでは 1 文字は 1 バイトまたは 2 バイトになる。日本語版 Microsoft Windows ならば ANSI コードページは CP932 になる。UNICODE と称するものは実際には UTF-16LE に似たもののことである(文字符号化方式を UNICODE と称するのは不適切かもしれないが、Microsoft が発行する文書で UNICODE と書かれているので、以下 UNICODE と記述する)。例えば CreateFile という API は実際には CreateFileA と CreateFileW がある。前者はファイル名を ANSI コードページの文字列で扱うのに対し、後者は UNICODE になる。多くの場合、ソースコードレベルでは CreateFileA や CreateFileW を直接扱うことはなく CreateFile と記述する。ヘッダファイルには UNICODE という名前のマクロが定義されているか否かで、CreateFile は CreateFileA または CreateFileW になる。文字列を表す型は TCHAR や LPTSTR であり、これも UNICODE というマクロで実際の型が決まる。ほとんどの文字列を引数にとる API と文字列をメンバにもつ構造体でこのような仕組みになっている。

Microsoft Windows NT とそれ以降の OS はファイル名に UNICODE を使うことができる。すなわち ANSI コードページには存在しない文字をファイル名に使うことができる。そのため引数が ANSI コードページな API ではファイルを正しく扱えない。引数が UNICODE な API を呼び出さない限り、ファイルを正しく扱えないのである。

7.Microsoft Windows における file name encoding の実装

Microsoft Windows では UTF-8 と file name encoding の関係はさらに複雑になる。ここまでは「file name encoding は open や stat、readdir のような API に使うことができる」という認識であった。GTK+-2.4 までは Microsoft Windows でもその認識で正しかった。Microsoft Windows ではどのバージョンも環境変数 G_FILENAME_ENCODING と G_BROKEN_FILENAMES には対応しておらず、GTK+-2.4 以前では file name encoding は ANSI コードページに固定されている。

しかし GTK+-2.6 以降は file name encoding の文字列を open や stat、readdir のような API に使うことができない。GTK+-2.4 以前のように file name encoding を ANSI コードページにしてしまうと、GtkFileChooser や g_dir_read_name で ANSI コードページにはない文字を含むファイル名を取得できなくなる。そこで GTK+-2.6 以降では file name encoding は UTF-8 として実装されている。GtkFileChooser や g_dir_read_name では可能ならば UNICODE 系の Microsoft Windows の API を呼び出し、UNICODE から UTF-8 への変換を行っている(不可能ならば ANSI コードページ系の Microsoft Windows の API を呼び出し、ANSI コードページから UTF-8 への変換を行っている)。

これに対応して GLib では g_open や g_stat などの API を新たに用意している。これらは g_open ならば open に、g_stat ならば stat と同じ働きをする。UNIX 系 OS ではこれらは単に名前が変わっただけとみなすことができる。Microsoft Windows では g_open や g_stat などの API の引数のファイル名はやはり file name encoding であり file name encoding の実体は UTF-8 なので、引数のファイル名は実際には UTF-8 になる。Microsoft Windows では例えば g_open は引数のファイル名を UNICODE に変換して wopen を呼び出す(UNICODE が利用できないならば引数のファイル名を ANSI コードページに変換して open を呼び出す)。g_open や g_stat などの API を利用すれば、

UNIX 系 OS でも Microsoft Windows でもソースコードの記述を一貫して同じにすることができる。g_open や g_stat などの API の説明は GLib Reference Manual の File Utilities にある。UNIX 系 OS であっても GTK+-2.6 以降ならば g_open や g_stat などの API を使うべきである。

8.file name encoding と"真の"ファイル名の文字符号化方式の関係

GLib Reference Manual の File Utilities にある API のうち g_chmod、g_access、g_creat、g_chdir は GLib-2.8 以降である。GLib-2.6 で chmod、access、creat、chdir を使うならば、file name encoding を ANSI コードページに変換しなければならない。file name encoding から ANSI コードページへの変換は g_locale_from_utf8 で、ANSI コードページから file name encoding への変換は g_locale_to_utf8 でおこなう。通常は file name encoding を UTF-8 とみなすことはできないし、ロケールの文字符号化方式が実際のファイル名の文字符号化方式と同一とは限らない。これは Microsoft Windows だけの特別な処理となる。また GLib-2.8 以降であっても GLib 以外の対応する API が GLib の File Utilities にはない API を呼ぶ時には、この変換が必要になる。

Glib-2.8 以降ならば g_win32_locale_filename_from_utf8 という API があるので、前述の g_locale_from_utf8 の代わりにこれを用いる。g_win32_locale_filename_from_utf8 は通常は g_locale_from_utf8 と同じだが、もしファイル名を変換できなくてかつ Microsoft Windows の UNICODE 系の API が使える時には、GetShortPathNameW で短いファイル名を取得して、取得できたならばそれを ANSI コードページの文字列で返す。

いずれにしても Microsoft Windows では ANSI コードページを引数にとる API のために、g_win32_locale_filename_from_utf8(GLib-2.8 以降)または g_locale_from_utf8(GLib-2.6)で変換する特別な処理が必要になる。

<table>
<tr><th>バージョン</th><th>file name encoding の実体</th><th>file name encoding から fopen などの API 呼び出し時の変換</th><th>File Utilities の API</th></tr>
<tr><td>2.4 以前</td><td>ANSI コードページ</td><td>不要</td><td>なし</td></tr>
<tr><td>2.6</td><td rowspan="2">UTF-8</td><td>g_locale_from_utf8</td><td>g_open、g_rename
g_mkdir、g_stat、g_lstat
g_unlink、g_remove
g_rmdir、g_fopen
g_freopen</td></tr>
<tr><td>2.8 以降</td><td>g_win32_locale_filename_from_utf8</td><td>g_chmod、g_access
g_creat、g_chdir</td></tr>
</table>

9.パスの区切り

パスの区切りに使われる文字は OS により異なる。UNIX 系 OS では「/」(2Fh)であるのに対して、Microsoft Windows では「\」(5Ch)である。GLib では G_DIR_SEPARATOR と G_DIR_SEPARATOR_S というマクロが定義されている。前者は数値としての区切り文字、後者は文字列としての区切り文字となる。例えば

```
#define G_DIR_SEPARATOR '/'
#define G_DIR_SEPARATOR_S "/"
```

となる。GLib-2.6 以降ならば G_IS_DIR_SEPARATOR(c)というマクロも利用できる。このマクロは UNIX 系 OS ならば c が G_DIR_SEPARATOR の時に真となる。しかし Microsoft Windows では特別な実装となっており、G_DIR_SEPARATOR(即ち「\」)または「/」で真となる。

GLib にはパスを扱う API としてファイル名を返す g_path_get_basename とディレクトリを返す g_path_get_dirname がある。これらの API の扱う文字列は file name encoding である。

Microsoft Windows では GLib-2.2.3 以降から g_path_get_basename と g_path_get_dirname は「\」と「/」の両方をパスの区切りとして扱う。上記のマクロ、G_IS_DIR_SEPARATOR が定義されるよりも前から「\」と「/」の両方をパスの区切りとして扱っている。

複数のパスを区切るための文字として G_SEARCHPATH_SEPARATOR および G_SEARCHPATH_SEPARATOR_S というマクロが定義されている。これは UNIX 系 OS では「:」(コロン)であり、Microsoft Windows では「;」(セミコロン)である。Microsoft Windows では「:」(コロン)はドライブ名に使われる。したがって、これらのマクロを使わずに複数のパスを区切るための文字を「:」(コロン)でハードコードすると移植性に問題が生じる。

10.5Ch 問題

Microsoft Windows では「\」(5Ch)をパスの区切りとするが、この 5Ch はマルチバイト文字の 2 バイト目になりうる。もし単純に文字列を 1 バイトづつ「\」(5Ch)と比較した場合、マルチバイト文字の 2 バイト目をパスの区切りとして処理してしまう可能性がある。例えば「表」という文字は CP932 では 95h 5Ch の 2 バイトになる。これをファイル名に含む「C:\doc\表 1.txt」(43h 3Ah 5Ch 64h 6Fh 63h 5Ch 95h 5Ch 31h 2Eh 74h 78h 74h)をディレクトリとファイル名に分けた時、43h 3Ah 5Ch 64h 6Fh 63h 5Ch 95h と 31h 2Eh 74h 78h 74h に分かれてしまう。

GLib-2.6 以降は file name encoding の実体は UTF-8 である。したがって 5Ch 問題は生じない。しかし GLib-2.4 以前は file name encoding の実体は ANSI コードページなので 5Ch 問題の可能性がある。

GLib-2.0.0 から GLib-2.4.8 までの GLib の実装を見ると、どのバージョンでも 5Ch 問題が生じる。単純に文字列を 1 バイトづつ「\」(5Ch)と比較しているので、GLib-2.4 以前の Microsoft Windows では g_path_get_basename と g_path_get_dirname は使えない。

11.iconv における 5Ch の扱い

SHIFT_JIS という文字符号化方式は JIS X 0201 と JIS X 0208 の 2 つの符号化文字集合(Coded Character Set)で構成される。ゆえに SHIFT_JIS では 5Ch は「\」ではなく「￥」となっている。iconv では SHIFT_JIS の 5Ch を UTF-8 に変換するときには C2h A5h(UNICODE の U+00A5、￥記号)に変換し、UTF-8 の 5Ch の変換ではエラーとなる。

CP932 でも 5Ch は「￥」となっている。しかし iconv は CP932 の 5Ch を UTF-8 に変換しても 5Ch のままにする。また UTF-8 の 5Ch を CP932 に変換する時も同様に 5Ch のままにする。したがって CP932 では SHIFT_JIS の時のようなパスの区切りが別の文字に変換されてしまう問題は発生しない。GLib は GetACP の返値が 932 のとき SHIFT_JIS ではなく CP932 を用いるので、パスの区切りが変換される問題は発生しない。

12.最後に

統計的な根拠のない私の偏見ではあるが、上級の UNIX 系 OS のユーザになるほどファイルやディレクトリの名前にスペースを含めないのと同様に US-ASCII 以外の文字をファイルやディレクトリ名には使わなくなる傾向にあると思う。US-ASCII の文字しか使わないならば、UTF-8 と file name encoding に間する問題は生じない。よって file name encoding のバグが露見しにくい。

しかし UNIX 系 OS が一般に普及すれば、ファイルやディレクトリの名前に US-ASCII 以外の文字を使うことが珍しくはなくなる。また GTK+は Microsoft Windows 版や Mac OS X 版もあり、GTK+を使ったプログラムはマルチプラットフォームとなりうる。せっかく GTK+を使ったのに、file

name encoding への対応が不十分であったために UNIX 系 OS でしか正常に動作しないというのでは、ガッカリである。

日本語を母国語とするプログラマは日本語環境のゆえにこの文字符号化方式の問題を理解しやすい立場にあると思う。私自身も自作のプログラム(Video maid、Text maid など)を通じて文字符号化方式の問題の手本となるように努力したい。

参考文献

[1] The GTK+ Team, ”GLib Reference Manual”, http://developer.gnome.org/doc/API/2.0/glib/index.html
[2] The GTK+ Team, ”GTK+ Reference Manual”, http://developer.gnome.org/doc/API/2.0/gtk/index.html
[3]Tony Graham, 乾和志, 海老塚徹, 関口正裕, “Unicode 標準入門”, 翔泳社
[4]Ken Lunde, 小松 章, 逆井 克己, “CJKV 日中韓越情報処理”, オライリー