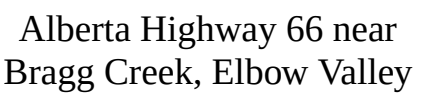
Why and how you ought to

Keep multibyte character support simple

EuroBSDCon, Beograd, September 25, 2016

Ingo Schwarze <schwarze@openbsd.org>

Canyon Campground below Lower Kananaskis Lake
and the Opal Range (2900-3000m)



Alberta Highway 66 near
Bragg Creek, Elbow Valley

Topic of this talk

Multibyte character support in the base system

- Multibyte character support in basic infrastructure
  like ksh(1), xterm(1), OpenSSH, man(1), libedit...
- LC_CTYPE support in BSD and POSIX utility programs,
  in particular the simplest ones like ls(1), ps(1), cut(1), wc(1)
- POSIX multibyte and wide character functions in the C library
- Multibyte character support in the kernel terminal driver?

Mount Nestor (2975m), the southern pillar of the Goat Range

Out of scope for this talk

Lower Kananaskis Lake (1680m)

- Internationalization in general
  That's a vast field and could only be covered in an overview talk.
  This talk explores specific technical details → limited scope.
- Not about general locale support — only about LC_CTYPE.
- Not about character encoding conversions.
  To convert files from one encoding to another,
  simply install the GNU iconv package.
- Not about typesetting.
  That cannot be done with C library utilities, and even Unicode is utterly inadequate to to express any non-trivial arrangement of glyphs, for example for non-trivial mathematical formulae or anything similar. Typesetting requires specialized software like TeX or groff, and in such contexts, character set handling is one of the points to be considered, but a relatively minor one, and in any case, for such software, it is completely irrelevant whether or not the base system offers any kind of multibyte character support or not.

It is not my intention to dismiss any real-world tasks as irrelevant, in particular not the handling of legacy non-UTF-8 multibyte encodings, which is a legitimate concern; i am merely studying what can reasonably be done with C library support in the base system without compromizing other goals like security, reliability, and usability, and what may better be left to specialized add-on software.

The myth of feasibility

- I suspect that many people think that as long as you implement carefully and use all standard facilities and interfaces as designed, complete multibyte character support can be done.
- At least i thought so before i set out on the quest i’m talking about.
- But it turned out that is not true. If you build support for arbitrary character set locales into the base system, there are several aspects with respect to which making things secure, reliable, and usable becomes outright impossible.
- As a first step, i will show some of these unsolvable issues.

A thunderstorm approaching Calgary, Center Street Bridge

Kananaskis Country, Alberta, Canada

Table of contents

An extreme example of breakage in the standard: write(1)

POSIX requires:

> “The following environment variable shall affect the execution of write: LC_CTYPE: Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files). If the recipient’s locale does not use an LC_CTYPE equivalent to the sender’s, the results are undefined.”

When the locales agree, POSIX requires:

> “Typing characters from LC_CTYPE classifications ‘print’ or ‘space’ shall cause those characters to be sent to the recipient’s terminal.”

~~Mount Warspite (2850m)~~
across the Kananaskis Lakes

Now as a matter of fact, for the sending program, there is no way to find out the recipient’s locale. By its basic design, the locale is part of the environment of each program, and it is essential for system security that without elevated privileges, no program can inspect the environment of other user’s programs.

So to satisfy the requirement for the case of matching locales, the standard effectively requires the write(1) program to unconditionally write all printable characters using the sender’s locale, no matter what the recipient’s locale may be.

write(1) implementation cannot be fixed

Locales mismatch → print garbage

- Even if the senders are well-intentioned and only send byte sequences they consider as printable characters in their own locale,
- the recipient's locale might interpret some of them as terminal control sequences
- may screw up the recipient's terminal state
- may display wrong and misleading information
- may even put the terminal into a state where it interpretes user input in a way different from what the recipient wants and reasonably expects.

Mist Mountain (3138m) from Lineham Creek

Impossible to reduce functionality to make it safe

For each and every byte sequence, there can be a locale in which it might represent a potentially dangerous control sequence, so no byte or byte sequence at all is safe to print to a terminal if you do not know the encoding.

Highway 22 near Bragg Creek,
Elbow River Valley

write(1) standard cannot be fixed

Standard effectively requires utterly unsafe behaviour

On a system providing arbitrary locales, there is no way how the standard could be improved, short of completely deleting the entire write(1) program.

Who uses write(1) anyway?

- People moved on to WhatsApp, didn’t they?
- But wait: wall(1) has the same problem.
- And shutdown(8) uses wall(1)!
- Worst time to screw up people’s terminals:
- Right before shutdown when they hurry to save their work...

That teaches us that even if many users use traditional low-level tools much less nowadays, they may still be more relevant in subtle ways than one might naively think.

Are important tools affected? →

Tough problems in basic tools: ssh(1)

An ssh(1) connection involves two locales

Goat Range (2700-2800m)
from Goat Pond

1. The locale set in the original shell on the client machine (client locale)
   Determines what can safely be displayed and how it must be encoded.
   It is already defined before the ssh(1) client program is even started.
2. The locale set in the remote shell on the server machine (server locale)
   Only this can influence what may get printed on the client
   to the terminal in which ssh(1) is run.

How is the server locale selected?

Lots of competing mechanisms for setting environment variables:

- Operating system defaults when starting new processes
- Variables set or unset by sshd(8) on the server when forking the login shell
- SSH initialization files, for example ~/.ssh/environment
- System wide and user specific shell initialization files, and so on
- ...

None of that host of possibilities depends on the locale used on the client side, or can even inspect the locale on the client side.

ssh(1) cannot be fixed either

- So we end up with a problem similar to the write(1) case:
- If the client side is using an arbitrary locale, the server cannot safely send any string in any encoding, not even plain US-ASCII.
- OpenSSH provides no way for the client to communicate the required locale to the server.
- And even if it could, there is no guarantee that that particular locale is available on the server.
- And even if there were a locale of the same name on the server, there is no guarantee that it is compatible with the client locale, because neither locale names nor the semantic of any locale except C and POSIX is standardized by POSIX.

Generic problem for any kind of inter-process communication

At the Goat Pond dam (1670m), Spray Lakes area

Partial mitigations for the ssh(1) problem

Chapman Bridge (ca. 1600m)
Elbow River Campground

- If you happen to know the default locale of the remote account you want to connect to and the same locale happens to be available on your client system, you can start a terminal using that locale on the client system before typing the ssh(1) command, and you are safe. But that's a special case.
- Besides, let me ask a question to the audience: Who has done that at least once in the past? Considering what the server locale was going to be, and start a matching terminal before typing the ssh command?
- Note that opening the connection first, then setting LC_CTYPE in the remote shell to whatever you need locally is not safe - the remote system may already print to your local terminal before you ever get to the shell prompt:
  - A banner even before authentication,
  - the motd(5),
  - and then the shell prompt itself...
- All that might already screw up your terminal, and in the worst case cause your terminal to misinterpret the input you type.

The OpenBSD way

Old Goat Mountain (3109m), Spray Lakes Reservoir

We made a drastic decision!

The OpenBSD base system supports
exactly two LC_CTYPE locales:

1. UTF-8
2. C = POSIX = US-ASCII

We don’t even support ISO-LATIN-1
any longer in the base system.

Isn’t that seriously inconvenient?

- Usability is not as bad as it may seem at first.
- If you get text in different encodings, it is very easy to install conversion tools from ports and simply convert the data once before using it.
- Besides, Unicode and UTF-8 support all languages.
- So even without relying on ports, the base system is still able to support all languages.

How does that help write(1)? →

Looking across Whiteman's Pond
to the Goat Range (ca. 2700m)

Benefits for write(1)

1. ASCII printable characters always safe to print on OpenBSD
   (UTF-8 ASCII-compatible: both encode ASCII the same way)
   That allows partial write(1) functionality:
   Allow passing ASCII only no matter what the two locales are.
2. It allows filtering out ASCII control bytes (C0 characters).
   Important because the escape character is dangerous.
   Possible because no UTF-8 sequence contains such a byte.
3. If the sender's terminal is set to UTF-8 and non-ASCII characters are actually typed, they can safely be filtered out (just in case the receiver's terminal is set to US-ASCII, which we cannot know). That's possible because UTF-8 is stateless, that is, codepoints can safely be deleted from the stream and it still remains a valid stream of characters (which would not be true for arbitrary locales).
4. If invalid bytes not forming UTF-8 occur in the input stream, they can safely be filtered out, allowing to recover from encoding errors. That's possible because after an encoding error, UTF-8 allows to find the beginning of the next character by simply looking for the next byte not having the most significant bit set or having the two most significant bits set.
   Consequently, the sending terminal can never become unusable, which might well happen when allowing arbitrary encodings.

Are there any downsides? →

Prices to pay in write(1)

OpenBSD write(1) now violates POSIX

Yet it does implement the maximal safe and useful and reasonable part of POSIX:
All printable ASCII characters, space characters, and the BEL are sent as typed.

- Locales are ignored.
- UTF-8 continuation bytes are silently ignored.
- ASCII control characters and UTF-8 start bytes are replaced with question marks.
- Consequently, if the sender uses UTF-8 or control characters, the recipient sees that something got lost and can ask the sender about the missing parts.

The implementation is extremely simple and robust

It doesn’t even need <wchar.h> or <locale.h>, neither setlocale(3) nor getwchar(3) nor mbtowc(3) nor anything like that. It gets away with elementary single-byte character handling functions like fgets(3), isprint(3), and putchar(3), which makes code review and maintenance a lot easier.

Spray Mountains (2700-2900m)
seen from the Kananaskis Lakes

Best practice for ssh(1)

- It is obvious that, if the server runs OpenBSD, which only supports the C/POSIX and UTF-8 locales, connecting to it becomes safe if you follow the simple rule to always use a UTF-8 enabled terminal to run ssh(1).
- Of course, that does not yet secure connections FROM OpenBSD systems: Connecting from OpenBSD to other operating systems is still dangerous because they might send text in arbitrary locales.
- So, the best practice i recommend is to:
  1. On all your servers that you may ever want to SSH into, no matter on which operating system, make sure the default system locale and all login locales are set to either C/POSIX or to UTF-8.
  2. Only ever run ssh(1) from UTF-8 enabled terminals.

Elpoca Mountain (3029m)
from the Smith-Dorrien Trail

While much of the stuff discussed here is subtle,
this is one simple pair of rules
i recommend that you remember.

What about xterm(1)? →

Benefits for xterm(1)

Upstream xterm(1) runs in ASCII mode by default

- Traditionally, that default was also used on OpenBSD.
- Bad idea: Some common UTF-8 characters are interpreted as control codes. For example, a stray German ‘ß’ will lock up your ASCII xterm(1).

OpenBSD 6.0 runs xterm(1) in UTF-8 mode by default

If you use a C/POSIX locale, even if you don’t intend to ever use UTF-8, that’s OK because a UTF-8 terminal handles ASCII output just fine.

In addition to that, the UTF-8 enabled terminal is obviously more resilient to UTF-8 accidentally sneaking in, in particular, but not only, for the case of running ssh(1) as explained above. Actually, even when fed garbage or unsupported encodings, a UTF-8 xterm(1) is more robust than an ASCII xterm(1) because the UTF-8 xterm(1) honours *fewer* terminal escape codes than the ASCII xterm(1). That may seem surprising at first because Unicode defines *more* control characters than ASCII does. But as explained on http://invisible-island.net/xterm/ctlseqs/ctlseqs.html xterm(1) never treats decoded multibyte characters as terminal control codes, so the ISO 6429 C1 control codes do not take effect in UTF-8 mode; but they do take effect in ASCII mode, even though they fall outside the scope of ASCII.

Caveats for xterm(1)

Do not use this non-standard default setting on any other system except OpenBSD.


Fairmont Range from Whiteman's Pond

- It only works because OpenBSD deliberately does not support any locales except UTF-8 and C/POSIX/ASCII.
- Terrible things will happen if you force the default to UTF-8 in this way on a system where people can opt into arbitrary locales that differ from UTF-8.
- On other operating systems except OpenBSD, there is no way in hell to make the interaction of locales with terminal controls truly safe.

The main goal of having UTF-8 xterms by default on OpenBSD is improving robustness.
But it also improves usability. If you usually run your shells inside xterm(1) in C/POSIX mode, there should be few visible changes for you.
But if you ever stumble upon a directory containing UTF-8 filenames, you can simply say

```
$ LC_CTYPE=en_US.UTF-8 ls
```

which would have given you garbage output in the past, and which just works in OpenBSD 6.0.

Benefits for pod2man(1) manuals

- Many perl manuals contain UTF-8.
- So do several ports manuals using perlpod(1) format.
- A few ports manuals contain ISO-LATIN-1: latex2man(1), a2ping(1), ...
  OpenBSD man(1), which is the mandoc implementation, silently converts that to UTF-8.
- So we enabled UTF-8 by default for pod2man(1) in OpenBSD,
  improving output for both UTF-8 and C/POSIX/ASCII users.
- Problem unsolvable on any system trying to support arbitrary locales,
  because man(1) must not print UTF-8 for users using a different locale.

Yellow Bellied Marmot near Lower Kananaskis Lake

Overview of implementation techniques

for small base system utilities

command	deci	parse	ins	sani	eval
	iddo	cvitgw	spw	inau	cwsp
rev	lc--	c-----	---	----	----
ksh	.---	c-----	---	----	----
tty(4)	----	c-----	---	----	-c--
write	----	c-----	---	-?p?	----
ypldap	----	c-----	---	-cc?	c---
cut -fd	l-ld	-v----	---	----	--s-
cut -cn	l-lc	--i---	---	----	c---
uniq -s	l-lc	--i---	---	----	c---
uniq -f	l-tf	---t--	s--	----	----
wc -m	le-c	---t--	s--	----	c---
colrm	l-t-	---t--	-pw	----	-w--
fold	lctb	c--t--	-pw	----	-w--
column	l-t-	---tG-	spw	----	-wS-
fmt	l-t-	---t--	spw	?ppp	-w--
ls	l-t-	---t-w	-pw	??pp	-w--
rs	le--	---t-w	-pw	??cc	-w--
ps	l-t-	---t-w	-pw	vrpp	-w--
ssh	l-t-	---t-w	-pw	vvcc	-w--
ul	l-g-	----g-	-pw	sspp	cw-p
man	l--l	p----w	--w	??pp	-w-p

utility functions

```
ls   int mbsprint(const char *mbs, int print)
rs   int mbsavis(char** outp, const char *mbs)
ps   int mbswprint(const char *, maxw, trail)
ssh  int vasnmprintf(c **, size_t, int *, fmt, va)
man  int preconv_encode(...)
```

```
decision making:
 i - initialization
     l - setlocale(3) called
     . - setlocale(3) called but essentially unused
 d - decision
     c - MB_CUR_MAX inspected in isu8cont()
     e - MB_CUR_MAX inspected before calling multibyte functions
     l - implicit in mblen(3)
     t - implicit in mbtowc(3)
     g - implicit in fgetwc(3)
 o - options deciding whether multibyte functions are used at all
     d - option to specify delimiter (may be UTF-8)
     b - option to count bytes (alternative is to count characters)
     c - option to count characters (alternative is to count bytes)
     f - option to count fields
     l - option to call setlocale(LC_CTYPE, "")
parsing:
 c - direct inspection with isu8cont()
 p - dedicated UTF-8 parser
 v - validate multibyte character with mblen(3)
 i - iterate multibyte characters with mblen(3)
 t - iterate multibyte characters with mbtowc(3)
 g - get wide characters with fgetwc(3)
 G - get command line argument with mbstowcs(3)
 w - wrapper to isolate UTF-8 handling from the main code
inspection:
 s - check for whitespace with iswspace(3) / iswblank(3)
 p - check printability with wcwidth(3)
 w - get display width with wcwidth(3)
sanitation:
 classes:
  i - invalid byte
  n - non-printable character
  a - printable ASCII character
  u - printable Unicode character
 actions:
  s - skip
  ? - replace with question mark
  v - encode with vis(3)
  r - replace with Unicode replacement character
  c - copy character
  p - print character
evaluation:
 c - count characters
 w - count display width for columnation and/or tabulation
 s - split strings with strstr(3)
 S - split strings with wcschr(3)
 p - print with putwchar(3)
```


Why this table? →

Why am i showing this table?

- Each line provides information about one program; some programs have very different modes, so some have two lines.
- Each column represents one particular implementation technique.
- My expectation, when showing this table, is not that anybody might understand the whole table during this talk. That is impossible, it encodes a huge amount of information in an extremely terse format. But there are several reasons why i do show the table anyway.
- First reason: It shows that the number of utilities we have to deal with is surprisingly small. You might expect that almost everything might need multibyte character handling. But this table only lists about three handfuls of utilities. Admittedly, we didn't fix all utilities yet, but most are done by now.
- Second reason: The table shows that the number of implementation techniques is surprisingly large. You might expect that there might basically be just one technique: Read a stream, assemble wide characters, process the characters, and write out the result. But it turns out that would be a bad approach almost everywhere, and different utilities have very different needs.
- Third reason: The table shows that so far, we did not not find a single pair of utilities that could be handled in exactly the same way. No line in the table agrees with any other line. Well, with one exception: cut -c and uniq -s can be implemented using exactly identical techniques - but the main technique used in that case appears literally nowhere else, and both utilities need quite different techniques when called with other options. So, basically, everything is different and nothing can be done schematically.
- Fourth reason, and i'll come back to that after explaining some of the techniques: Those techniques that people would probably have expected to be ubiquitous barely appear at all. For example, fgetwc(3) and fputwc(3) appear in one out of 16 cases, and fgetws(3), fputws(3), *wprintf(3), *wscanf(3), mbrtowc(3), wcrtomb(3), and wmem*(3) don't occur at all.
- At this point, some of you will probably wonder whether i screwed up the title of my talk. "Why and how you ought to keep multibyte character support simple" - but now the guy is saying there is a large number of different techniques and everything differs from everything else? That doesn't sound simple at all!
- The point is: While the number of techniques is indeed not all that small, every single technique is very elementary, so the code of every individual utility does remain very short and simple. Much simpler, in fact, than it would become if you tried to apply one and the same general-purpose coding scheme everywhere.

Technique 1: pure isu8cont()

rev(1) - reverse characters in each line

requirements: no need for character properties, only need to know where chars start
hence, no need for character decoding
not even any need for character validation

solution: isu8cont() one-liner by Ted Unangst <tedu@openbsd.org>:
Does this byte continue a character?

```
int isu8cont(unsigned char c)
{ return MB_CUR_MAX > 1 && (c & (0x80 | 0x40)) == 0x80; }
```

setlocale(3) to set up MB_CUR_MAX

algorithm: read lines into a char[] buffer, skip newlines
loop over characters by looping over bytes and skipping on isu8cont()
then just copy the multibyte sequences without ever decoding them

This technique is very robust and never fails.
Of course,
it only works for UTF-8 only systems;
no similar technique is possible
for arbitrary encodings.

Smith-Dorrien Trail (1650m)
with East End of Rundle (2550m)

Technique 1: pure isu8cont()

The alternative: rev(1) on FreeBSD and NetBSD

Dust on the Smith-Dorrien Trail
with Mount Sparrowhawk (3121m)

- setlocale(3) used in the same way
  loop over lines with fgetwln(3)
  store wide character strings
  then print wide characters in reverse order
- very fragile, dies on the first encoding error
- even worse: all known implementations of fgetwln(3) were buggy and return success on encoding errors, effectively converting all encoding errors into newline characters → silently gave wrong results

Summary regarding isu8cont()

- pure isu8cont() is the right move when you need no validation and no character properties
- another example of pure isu8cont() is write(1) mentioned above

Technique 1: pure isu8cont(1) for ksh(1)

It can also be used as a quick and dirty stopgap for complex programs

for example OpenBSD ksh(1) emacs input mode

Elbow Falls

- Make sure that moving left and right
  can only move by whole characters,
  not into the middle of a character.
- Make sure that deleting characters
  can only delete characters whole,
  not individual bytes out of characters.
- Improve all functions involving words
  by allowing non-ASCII characters
  to be part of words.
- Allow insertion of non-ASCII characters
  without screwing up the display,
  by backing up to the start byte
  after inserting a continuation byte,
  and starting to re-print there.

Technique 1: pure isu8cont(1) in the kernel

A quick and dirty stopgap for the tty driver

Calgary, Bow Tower

- For OXTABS and sysctl(3) KERN_TTY_INFO, sys/kern/tty.c wants to calculate display widths.
- We don’t want wcwidth(1) tables in the kernel, so double- and zero-width characters will remain wrong.
- But let’s at least handle the common case of single-width multibyte characters correctly:

```
int
ttyoutput(int c, struct tty *tp)
{
        int col = 0;

        /* lots of code deleted */
        switch (CCLASS(c)) {
        case ORDINARY:
                if (!isu8cont(c))
                        ++col;
                break;
        }
}
```

Technique 2: pure mblen(3)

cut -d — select delimited fields out of lines

Tsuu T’ina land near Calgary

requirements: no need for character properties,
but need validation

solution:

```
case 'd':
        dlen = mblen(optarg, MB_CUR_MAX);
        if (dlen == -1)
                usage();
        memcpy(dchar, optarg, dlen);
        dchar[dlen] = '\0';
```

Alternatives

FreeBSD: uses mbrtowc(3), which is also possible

NetBSD: no multibyte support

Technique 3: iteration with pure mblen(3)

cut(1) -c — select by character count

Again no need for character properties, but need validation:

```
while(*cp != '\0') {
        len = mblen(cp, MB_CUR_MAX);
        if (len == -1)
                /* Handle encoding error, at least set len. */;
        /* Do something with the character. */
        cp += len;
}
```

Decision in OpenBSD: treat each invalid byte as one character and keep going.

Lineham Ridge (2700m) from the Highwood Valley (1880m)

Alternative

FreeBSD and NetBSD:

Use getwc(3)
and error out of the file
on the first encoding error.

cut(1) in FreeBSD

- FreeBSD: uses fgetln(3) and mbrlen(3)/mbrtowc(3).
- That is slightly inconsistent. Even though POSIX 2016-TC2 now requires that L'\n' be encoded as 0x000a in wchar_t, that doesn't imply that an arbitrary locale must encode it as the single byte 0x0a in a multibyte char * string, or as any single byte at all.
- In any case, an encoding error causes the rest of the file to be lost.
- mbstate is never reset, encoding errors in earlier files may compromise decoding of later files.

Looks like a pack of well-fed dragons...

Johnson Trail (Hwy. 532)

- This teaches that it's very easy to write code that looks perfectly general on first sight, but turns out to actually be full of subtle issues on closer inspection.
- As OpenBSD prefers correctness, security, and usability over featurism, we believe that it's advantageous to sacrifice full generality up front and allowing a simpler, more powerful, and less fragile implementation for UTF-8 and ASCII only.

Technique 4: iteration with mbtowc(3)

This one is most often needed in practice.

Many summits: Spray Mountains
from the Smith-Dorrien Trail

```
char    *mbs;   /* Multibyte string (input). */
int      len;   /* Encoded length in bytes. */

wchar_t  wc;    /* Wide character (decoded). */
int      width; /* Display width in terminal columns. */

for (mbs = INPUT; *mbs != '\0'; mbs += len) {
        HANDLE_SPECIFIC_BYTES(*mbs);  /* Optional. */
        len = mbtowc(&wc, mbs, MB_CUR_MAX);
        if (len == -1) {
                /* Encoding error, reset state: */
                mbtowc(NULL, NULL, MB_CUR_MAX);
                /* After handling an invalid byte, retry with the next one. */
                len = 1;
                HANDLE_INVALID_BYTE(*mbs);  /* Optional. */
                wc = L'?';      /* e.g. fmt, ls, rs, uniq */
                wc = L' ';      /* e.g. wc */
                width = 1;      /* e.g. column, colrm, fmt, ls, rs */
                width = -1;     /* e.g. ssh */
        } else {
                width = wcwidth(wc);
                if (width == -1) {
                        HANDLE_NONPRINTABLE_CHARACTER(wc);  /* Optional. */
                        width = 1;  /* Usually. */
                }
        }
        HANDLE_CHARACTER(wc, width);
}
```


Modularize? →

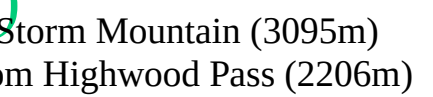
Technique 5: utf8.c utility files

- Advantage: isolate all multibyte = UTF-8 handling in one file
- Avoid encumbering the main code
- Not always possible, sometimes not even desirable, in particular if the main code doesn't do much except character handling in the first place, like in cut(1) or fmt(1)
- Typical tasks: parsing, validation, sanitation, output
- Typically uses technique 4, iteration with mbtowc(3)
- Sanition concerns invalid byte sequences and non-printable characters
- Sanitation options are passthrough, skip, replace with question marks or UTF-8 replacement characters, or vis(3)
- Examples: ls(1), ps(1), rs(1), OpenSSH
- All have subtly different requirements, in particular regarding sanitation, width measurement, width limitation, and output disposition
- Hence, it was not yet possible to design a set of standard functions, but we still hope more experience might allow to do so in the future.

Looking from the Three Sisters Dam (1710m)
to the northern Goat Range (2730m)

POSIX standard functions? →

Technique 6: iteration with fgetwc(3)


Storm Mountain (3095m)
from Highwood Pass (2206m)

- Only program so far too complex for these techniques: ul(1)
- I implemented and tested a version using techniques 4 and 5,
  iteration with mbtowc(3) and wcwidth(3) in utf8.c,
  but it wasn't simple at all, so it was never committed.
- The reason why it wasn't simple: It does all kinds
  of string manipulation, almost like an editor:
  splitting, joining; deleting, inserting, transforming characters
- What i did commit was a version doing the full
  char * to wchar_t * to char * double conversion.
- In part inspired by the FreeBSD version which is in turn
  based on Bruno Haible's work in util-linux,
  but not sharing any UTF-8 code with either version.

Examples of problems with the FreeBSD version of ul(1)

- Errors out on the first encoding error — can't be helped when surporting arbitrary encodings.
- Backspace backs up one column position, but should backup one character.
- Always treats _\b_ as underlined, never as bold.
- Fails to move the rest of the buffer right when _ later gets overlaid with a double-width char.

Techniques to avoid

- *r*() functions like mbRtowc(3)
  Don’t use them unless you really need multithreading.
  They are considerably harder to use correctly than mbtowc(3).
- fgetws(3) or fgetwln(3)
  Don’t use them unless you must support arbitrary locales.
  getline(3) allows much better error handling and isn’t harder to use.
  (And even for arbitrary locales, read(2) + mbtowc(3) is an option.)
- *towcs() functions like mbstowcs(3)
  Iterating with mbtowc(3) allows much better error handling
  and needs only marginally more code (typically a dozen lines of code).

Biking the Crowchild Trail, Calgary

Some people recommend
to always use the most complicated functions
because they work in all circumstances.

I don’t.

Where simpler functions suffice, they are easier to use and cause less bugs. And the simpler functions themselves may be less buggy, too.

What about library quality? →

Library quality

- In general, the BSD C libraries are of good quality:
  solid code that has been scoured for bugs for decades.
- Multibyte and wide character code is no longer exactly young,
  but younger than much other code, and much more buggy.
- In the following, i'm showing various examples from OpenBSD.
  Other BSD implementations sometimes differ in detail,
  but my impression is that quality is similar in all three systems.
- All bugs found by chance, no complete audit yet!

Going from the Kananaskis Lakes toward Highwoood Pass

What about the most basic functions? →

Examples of bugs in conversion functions

- mbtowc(3) neglected to set errno(2) to EILSEQ
  when given an incomplete character (fixed Feb 2016)
- mbrtowc(3) accepted some invalid UTF-8 sequences
  and silently produced invalid code points above U+10FFFF (fixed Sep 2015)
- wcrtomb(3) accepted code points above U+10FFFF
  and silently produced invalid multibyte sequences (fixed Sep 2015)
- wcrtomb(3) accepted UTF-16 surrogates in UTF-8 mode
  and silently produced invalid multibyte sequences (fixed Oct 2015)

Little Elbow River (ca. 1600m) washed out by the 2013 flood, with Mount Glasgow (2935m)

Bugs in libraries: examples in standard I/O

- fgetwc(3) didn't set the error indicator for encoding errors (fixed Dec 2015)
- fputwc(3) didn't set the error indicator for invalid characters (fixed Jan 2016)
  The Austin Group thinks that even the C standard itself is buggy here.
- fgetws(3) discarded any characters read and reported bogus EOF when errno happened to be EILSEQ upon entry and the file ended without a terminating L'\n' character (fixed Jan 2016)
- fgetwln(3) ignored most encoding errors and
  sometimes returned partial lines truncated at random places (fixed Aug 2016)
- printf(3) %ls destroyed all file flags on encoding errors,
  making the file permanently unreadable and unwriteable (fixed Jan 2016)
- printf(3) silently treated encoding errors in the format string
  as the end of the format string (fixed Jan 2016)
- printf(3) accessed a NULL pointer when out of memory or on encoding errors (fixed Jan 2016)

The scree of the Rock Glacier (ca. 2100m), Pocaterra Valley,
and the Elpoca Mountain (3029m)

More examples? →

Various other errors in the C library

- The character property tables contained no data whatsoever for characters in the range U+FF00 to U+10FFFF. (fixed Oct 2015)
- Due to a bug in mklocale(1), character type and width data was wrong for many exotic characters designating numbers. (fixed May 2016)
- The C library code parsing character property tables contained out of boundary memory access for corrupt input files. (fixed Oct 2015)

Repair of the Highwood River (1870m) after the 2013 flood below the Elk Range (2750m)

Examples of bugs in libedit

- el_wgetc(3), el_wgets(3) etc. sometimes discarded valid bytes after reading invalid bytes
- el_getc(3) silently converted non-ASCII Unicode characters into bogus bytes
- el_getc(3) didn't set errno(2) for out-of-range errors
- el_getc(3) didn't set the return argument to the NUL byte on read errors
- Several functions reading characters broken on systems where wchar_t doesn't use UCS-4.

All these bugs were found during a partial audit and fixed in March 2016.

Crumbling rock on the south face of Storm Mountain (3095m)

Examples of bugs in various programs

- mandoc(1) violated ISO C99 by mixing putchar(3) and putwchar(3) on the same stream, resulting in corrupt output on glibc (fixed July and September 2016)
- mandoc(1) accepted UTF-16 surrogates in \[uXXXX] escapes and silently produced invalid UTF-8 output (fixed Oct 2015)
- mandoc(1) failed to apply bold and italic markup to non-ASCII characters (fixed Oct 2015)
- tmux(1) contained wrong display widths for various characters in its internal width tables (fixed Nov 2015)
- ypldap(8) contained buggy hand-rolled UTF-8 validation code that failed to actually validate but instead caused buffer overruns and loss of input data for invalid input (fixed Apr 2016)

Even the tiny Smuts Creek (ca. 1900m) was washed out and needed repair.

Background: Spray Mountains (up to 3400m)

The situation in OpenSSH

- Input and output streams are treated as narrow throughout.
- In most places, incoming text data is treated as opaque byte strings, simply passing it through, not even trying to validate or decode.
- Where text data is interpreted, the code does not restrict itself to UTF-8, but attempts to support arbitrary encodings, though not always in full generality.
- Informational and diagnostic messages are written in ASCII throughout, which is compatible with many, but not with all encodings.
- In scp(1) and sftp(1), untrusted text data sent from the server - for example file and directory names - could silently screw up terminal settings on the client host in addition to wrong data being displayed. Part of that was fixed in May 2016 using validation and sanitation techniques. Some fixes could not yet be committed because part of the output is produced in signal handlers.
- For exotic encodings, several unknown bugs likely exist.
- For maximum security, make sur both endpoints run OpenBSD (to avoid exposure to arbitrary locales) and set LC_CTYPE to UTF-8 on both sides.
- sftp(1) libedit usage needs review with respect to multibyte character handling.
- Overall, auditing has barely begun.

Conclusions

- Multibyte character handling code in full generality is huge, complicated, buggy, must error out on the slightest problem, and using arbitrary encodings is never fully secure.
- Consequently, providing full multibyte support in a general-purpose POSIX C library is bad for usability, correctness, and security.
- Instead, better leave full generality where it belongs: In dedicated conversion and text processing software.
- Supporting UTF-8 only allows good usability, simplicity, and hence a better chance for correctness, reliability, and security.
- No silver bullet found yet to solve all implementation tasks in one unified way, but every practical task encoutered so far allowed a simple solution.
- A full toolbox may require up to ten different simple implementation techniques, each one adapted to a specific set of requirements.
- Look at the source code of the OpenBSD utilities mentioned here for guidance how to solve similar tasks.

East End of Rundle (2550m) seen from Canmore

Work to do in OpenBSD

- Small utilities almost done, but a few remain: lam(1), pr(1), talk(1), tr(1), ...
- Continue audit: libc, libedit
- POSIX regular expression library, fnmatch(3), glob(3)
- Work was barely started: ksh(1), OpenSSH
- Work was not yet started: vi(1), mg(1)
- Unknown status, maybe not much to do: libcurses, less(1), ...

Outlook from The Hump (2020m)
along Johnson Trail (Hwy. 532)
across the foothills towards the prairies

Thanks!

- The OpenBSD foundation provided financial support for part of my time. No way to make so much progress without that — but without the contributions by lots and lots of other developers, nothing would have been achieved at all:
- **Marc Espie**: initial import of partial multibyte handling code
- Stefan Sperling: import of partial Citrus code; contributed to many of the ideas explained here; joint work, patch reviews, bug reports, many useful discussions
- Ted Unangst: developed many of the ideas explained here; many patch reviews
- **Anthony Bentley**: contributed to many of the ideas explained here; joint work, bugfixes, patch reviews, bug reports, many useful discussions
- **Andrew Fresh**: gen_ctype_utf8.pl author and maintainer; bug reports and feedback on some patches
- Martijn van Duren: fix rev(1), wall(1), write(1); many code reviews for libedit; some initial ideas and lots of feedback for scp(1) and sftp(1)
- Sébastien Marie: joint work, some bugfixes, patch reviews, useful discussions
- Vadim Zhukov: review and meticulously refine the TODO list; feedback concerning some of the ideas explained here; some patch reviews
- Todd Miller: large numbers of patch reviews in libc and utilities

Thanks! (2)

- Theo de Raadt: support for the OpenSSH, stdio, and citrus patches; support for the general direction with small utilities; a few patches and patch reviews
- Dmitrij Czarkoff: many code reviews for libedit and patch reviews for utilities
- Damien Miller: patch to allow UTF-8 in ssh(1) banners
  and much support working on OpenSSH code in general
- Nicholas Marriott: UTF-8 work in tmux(1);
  feedback concerning some of the ideas explained here
- Christian Weisgerber: feedback concerning some of the ideas explained here; important suggestion and patch review for xterm(1)
- Peter Hessler: important contributions to some of the ideas explained here

Additional patches, patch reviews, help and feedback from:

Darren Tucker, Eric Faurot, Giannis Tsaraias, Jason McIntyre, Jérémie Courrèges-Anglas, Jonathan Gray, Martin Natano, Martin Pieuchot, Masahiko YASUOKA, Masao UEBAYASHI, Matthieu Herrb, Philip Guenther, Stuart Henderson, Theo Bühler, Tobias Stöckmann (OpenBSD), Andrey Chernov (FreeBSD), Christos Zoulas (NetBSD), Svyatoslav Mishyn (Void Linux), Christian Heckendorf, Frederic Nowak, Matthew Martin, ...