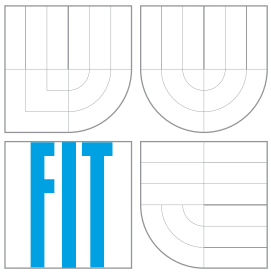
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

ROZŠÍŘENÍ PODPORY UNICODE PRO ZÁKLADNÍ GNU NÁSTROJE

ENHANCED UNICODE SUPPORT FOR GNU COMMAND LINE UTILITIES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — ONDREJ OPRALA
AUTHOR

VEDOUCÍ PRÁCE — Dr. Ing. PETR PERINGER
SUPERVISOR

BRNO 2015

Abstrakt

Tato práce řeší problém chybějící podpory pro správné zpracováni Unicode vstupu v programech projektu coreutils. Podpora byla implementována pro programy cut, expand, fmt, fold, paste a unexpand. Implementace byla provedena s využitím knihoven libunistring a gnulib. Programy byly řádně otestovány a výkonnostní testy potvrdily že výkon programů je porovnatelný nebo i lepší než u původní implementace.

Abstract

This thesis solves the problem of missing support for proper handling of Unicode input in the utilities of the coreutils project. Support was implemented for utilities cut, expand, fmt, fold, paste and unexpand. The implementation was done using the libunistring and gnulib libraries. Programs were properly tested and performance tests proved that performance is comparable or superior to the original implementation.

Klíčová slova

multibajt, Unicode, coreutils, libunistring, GNU/Linux, Open-source, internacionalizace, lokalizace

Keywords

multibyte, Unicode, coreutils, libunistring, GNU/Linux, Open-source, internationalization, localization

Citace

Ondrej Oprala: Enhanced UNICODE Support for GNU Command Line Utilities, bakalářská práce, Brno, FIT VUT v Brně, 2015

Enhanced UNICODE Support for GNU Command Line Utilities

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval sám pod vedením Dr. Ing. Petra Peringra.

.....
Ondrej Oprala
May 19, 2015

Poděkování

Rád bych poděkoval Dr. Ing. Petrovi Peringrovi a Ing. Ondřejovi Vašíkovi za jejich čas, postřehy a pomoc při vytváření této práce. Taktéž bych rád poděkoval Bernhardovi Voelkerovi z OpenSUSE a Pádraigovi Bradymu B.Eng. z projektu GNU za revize kódu, názory a obecné rady.

Contents

Chapter 1

Introduction

In the contemporary era, where the computer industry is experiencing a rapid growth, computers are no longer just a room-sized university apparatus for a chosen few, but can be found almost anywhere. Be it schools, medicine, government processes, home appliances or indeed any aspect of life in general, computers are present in all our daily lives, either directly or indirectly.

Commonly, the first personal computers were only able to accept English characters as input, limiting their applicability to only some regions of the world, regarding real-life, non-scientific usage. With the advent of Internet connectivity in common households, computers have since also been queried with more personal tasks than scientific or business computing. E-mails, social networks, forums, and generally any kind of text processing could now be done by computers, although, initially with severe limitations. With all these concepts reinforcing inter-personal communication and information sharing, the need to properly represent other alphabets and characters has arisen. E-mail and chat clients, web browsers, text editors and document viewers were all expected to be able to process, store and often modify such content in a consistent manner.

Out of this need, several standards to represent special characters appeared. Most extended the standard US encoding with other suitable symbols. Many only addressed a certain subset of characters, making them incompatible with other such standards. Eventually, the number of encoding standards in existence made it more and more difficult for application writers to keep up and support them all. In 1987, Joe Becker of Xerox [1] pondered the idea of uniting these standards into one, uniting all of the world's symbols into a single character set. Thus, the Unicode standard was created, successfully phasing out most other encodings and being implemented in programs world-wide. The standard is continually under development until this day and consistently getting used more and more. Nowadays, proper access to information in the users' native language is not considered to be anything out of the ordinary. Yet, core applications and utilities created in the era of first personal computers are still often unable to cope with non-US letters and characters.

As a citizen of a non-English speaking country, I've also encountered problems with the processing of my native language. This fact, combined with my experience with low-level GNU/Linux tools made this thesis an ideal choice for me. As the package of utilities chosen is present on almost all GNU/Linux and UNIX systems, I think it's important it provides consistent support for the Unicode standard. My goal in this thesis is to analyze the current status of the coreutils' package ability to handle text containing international characters. Furthermore, I am to provide an implementation of the standard for at least 5 utilities, using an existing library approved by the current project maintainers. I'll then test and

benchmark the utilities against the existing support, if any. The last step is sending these patches to the current upstream maintainers.

In Chapter 2, I'll talk more about the evolution of Unicode, but also other encodings, Unicode's history, its overall goals, described some important details of the standard and associated issues. Chapter 3 summarizes the existing implementation, mentions its drawbacks and offers an alternative agreed upon by the coreutils maintainers. Chapter 4 describes my implementation efforts in detail, as well as the required configuration of the package. Further, it describes testing, picks out interesting or diverse test cases from the testsuite and shows the results of benchmarking, where appropriate.

Chapter 2

Character encodings

This chapter contains a brief history and evolution of text encoding used in personal computers, and most importantly outlines the Unicode standard's history [1], rationale and development [14]. It also covers encodings defined by the standard [7], as well as deprecated encodings, their usage, differences, advantages, and disadvantages [9], [10]. Separate subsections are also dedicated to sorting and ordering, combining characters, surrogate pairs and also planes and blocks. Some basic terms are also explained within the chapter, where deemed appropriate. However, this chapter is not meant to be an exhaustive description of Unicode, its applications or usage. I have used also used information from the book [22] for this chapter.

2.1 History of character encoding

This section summarizes, in chronological order of their first appearance, the most common standards for representation of characters on computers. It also briefly introduces Unicode's development over time.

2.1.1 ASCII

The abbreviation ASCII stands for American Standard Code for Information Interchange. The standard defined a table of 128 characters and their codepoints to be used as their standard representation in computers [2] and it's been a de facto standard of computer communication for decades. It has practically been a standard character encoding since teletype machines were in use [8]. However, it became widely-used in personal computers only after 1981, when IBM decided to migrate from their own EBCDIC encoding to ASCII.

2.1.2 ISO-8859

The ISO-8859 family of standards of 8-bit encodings, extending the ASCII character set, often called extended ASCII. This led to a widespread notion that there is a single extended ASCII table, whereas the term actually refers to all ASCII-based 8-bit character sets. The standard comprises of 15 different parts, describing different encodings and associated character sets. The ISO-8859 scheme itself was actually an attempt to consolidate more cluttered industry standards at the time [9]. Similar to ASCII, the encodings contain a mixture of characters for historical reasons, but each one to serve a different Latin language. As mentioned before, Unicode was designed to begin with the ISO-8859-1 as its subset [10].

Although its share is constantly dropping in favor of UTF-8 (see 2.2.2), ISO-8859-1 is still the second most common encoding of web pages. As of late 2014, its share dropped under 10% for the first time. See figure 2.1 for details.

2.1.3 Unicode

By definition, Unicode is the universal character encoding, maintained by the Unicode Consortium. This encoding standard provides the basis for processing, storage and interchange of text data in any language in all modern software and information technology protocols. The "Uni" part of the name stands for "universal", "uniform" and "unique" [7] [1].

The initial idea behind Unicode was to consolidate all contemporary languages and their associated alphabets into a single encoding. The initial design was concieved by Joe Becker of Xerox [1] expected that each character would have a static 16-bit codepoint and dismissed the idea of variable length encoding. The Unicode88 document assumed that 16-bits (65536 values) were more than enough to represent all the world's modern characters, including Japanese, Chinese and Korean. Originally, ancient scripts and symbols weren't considered for inclusion [1] and were instead considered "better candidates for private-use registration than congesting the public list of generally useful Unicodes."

The first 256 characters of the Unicode table are identical to the ISO-8859-1 (see 2.1.2) standard, also known as latin-1. However, the extended ASCII part is encoded in two bytes, thus a latin-1 string isn't automatically a valid UTF-8 string. ASCII being a subset of the ISO-8859 family of standards, they are both subsets of the Unicode character table. Additionally, apart from the proper ASCII subset, all other characters are encoded as a collection of 2 to 4 bytes in the range of hexadecimal values 0x80 to 0xfd, so ASCII characters (in the range 0x00 to 0x7f) can never appear in the "middle" of other characters. This is also the main reason why UTF-encoded ASCII strings are almost as fast to process as if they were in their original ASCII encoding.

For Unicode 2.0, it became apparent that 16-bits would not be enough, and the standard introduced surrogate characters (see 2.2.9) to represent characters outside of the range of the 16-bits. Unicode is currently at version 7.0.0. The latest version added over 2834 characters, including support for lesser-used languages, currency symbols, historic scripts and pictographic symbols such as emoji [7]. The next version of the Unicode standard will be version 8.0. The new version should bring color modifiers for the recently added emoji characters. The enhancement request was delivered to the Unicode standard committee by the Google and Apple companies, claiming that the current Unicode character set is not multicultural enough [14].

2.2 Unicode architecture

The Unicode Standard's ultimate goal is to provide a universal encoding that encompasses every known character, be it a current or ancient one. However, it is not just one big table of characters. It provides different encodings for different use-cases and provides ways to combine characters together to create new ones.

In this section, I'll summarize current Unicode encodings, their differences, how they represent characters, their shortcomings and common usage. I'll also describe some issues not related only to Unicode encodings. Further subsections cover how the Unicode table is divided and some basic terms regarding sorting, combining characters and surrogate pairs

are defined. The section ends with a short summary of deprecated encodings and reasons for their deprecation.

2.2.1 Basic terms

This section explains some basic terms the reader can come across throughout the thesis, as they are defined in the Unicode standard.

Codepoint is a numerical representation for a character. It uniquely references a character.

Character is a single abstract independent entity in the Unicode table. A character may be a letter, a punctuation mark or a an entity with no visual representation by itself.

Glyph is the graphical representation of a character. The rendering of a glyph is not handled by the Unicode standard.

Script is a set of letters that are used together in writing languages.

Font is a repertoire of glyphs, similarly as a script is a repertoire of letters.

Internationalization and localization

Internationalization and localization are two terms often used interchangeably, although their meaning is slightly different.

Internationalization is the process of designing software so that it can be localized for various user communities without having to change or recompile the executable code [9].

Localization is the process of converting an application for use by a new user community. Localization involves not just translating any user-visible text, but also altering things like pictures, color schemes, window layouts, and number, date, and time formats according to the cultural conventions of the new user community [9].

Very often, Internationalization is abbreviated to I18n and localization to L10n, the numbers 18 and 10 denoting the length of characters between the first and the last letter of the respective word.

2.2.2 UTF-8

The UTF-8 encoding was not in the initial proposal, which only had a fixed 16-bit encoding in mind and actually dismissed the variable-length encoding as too cumbersome. UTF-8 was designed to be used in places where a person would usually use an 8-bit encoding. This is the main reason it's used on most GNU/Linux and UNIX systems, since other locales are traditionally 8-bit. A character may be represented by 1 to 4 bytes and the actual character byte-length is determined algorithmically. This has an important implication when searching for a character. The maximum length implies, that the algorithm always has to search at most 3 bytes before or after the examined character [9].

Some characters may have several representations in the Unicode table as they can be composed of several characters (for example accented letters, see 2.2.8 for more details). The standard also specifies that the shortest existing representation of a character must be used. The non-shortest representations of a symbol are called overlong encodings and are no longer legal.

UTF-8 is being aggressively adopted on the web and in text transmission over the network. Figure 2.1 shows this continuous trend. Its other significant usage is on GNU/Linux systems, where it has become a de facto standard for locale encodings.

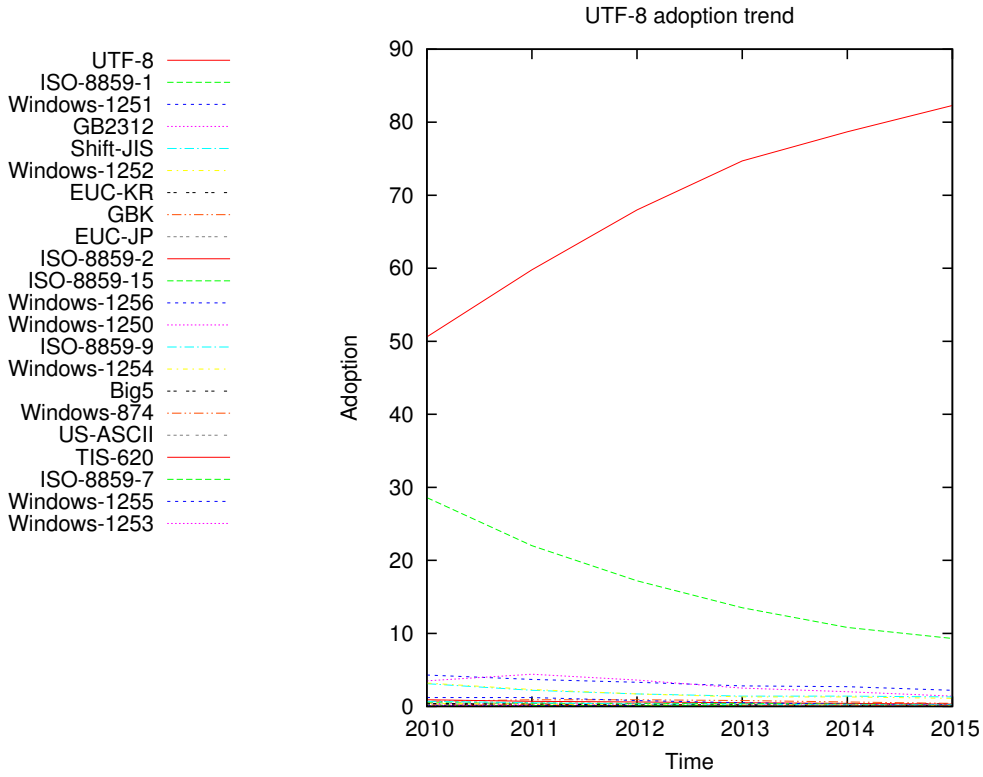


Figure 2.1: UTF-8 adoption trend

Overlong UTF-8 sequences

Initial Unicode standards permitted 5 and 6 byte UTF-8 sequences. These were possible by using the non-shortest character representation from the supplemental planes (see 2.2.7), which were also legal at the time. The use of five and six byte UTF-8 characters is now illegal, and a reasonable up-to-date parser should reject them.

An often implemented exception to the overlong rule is the representation of the NUL character (0x00) as 0xC0 0x80. Programmers sometimes embed this sequence in strings instead of the shorter version, since languages such as C and C++ have trouble incorporating a NUL character in a string without considering it a delimiter [25]. Table 2.1 practically summarizes legal UTF-8 sequences.

2.2.3 UTF-16

UTF-16 is a variable length encoding. Is uses either one or two 16-bit words to represent a symbol from the Unicode codeset. The symbols outside of the Basic Multilingual Plane (BMP) are addressed with the use of surrogate pairs (see 2.2.9).

Please note that this is not the encoding initially specified by the Unicode88 paper, although very similar. The initial encoding was called UCS-2 and was only able to address

Table 2.1: Legal UTF-8 sequences

UCS Code (Hex)	Legal UTF-8 Values (Hex)
00-7F	00-7F
80-7FF	C2-DF 80-BF
800-FFF	E0 A0-BF 80-BF
1000-FFFF	E1-EF 80-BF 80-BF
10000-3FFFF	F0 90-BF 80-BF 80-BF
40000-FFFFFF	F1-F3 80-BF 80-BF 80-BF
100000-10FFFF	F4 80-8F 80-BF 80-BF
200000-3FFFFFF	too large
04000000-7FFFFFFF	too large

the Basic Multilingual Plane (see 2.2.7), the first 16 bits. UCS-2 was most commonly used on Windows NT systems, before Microsoft's decision to switch to UTF-16.

Since UTF-16 is word-oriented, it requires a way to specify the byte order. One way to achieve unambiguity is to use the BOM (see 2.2.11) mark. The other approach is explicitly specifying the endianness in the form specified by the Unicode standard. In the case of UTF-16, this is called the UTF-16LE or UTF-16BE, LE and BE standing for Little Endian and Big Endian, respectively (for more on endianness, see 2.2.11).

UTF-16's most prominent usage is in the Java and .NET family of languages, exclusively in Python2.0's *unicode* class and on modern Windows systems in general (see 2.3.1).

2.2.4 UTF-32

UTF-32 is a fixed-length encoding representing each Unicode symbol with a 32-bit value. UTF-32's biggest problem is its size wastefulness. Since the vast majority of symbols found in common text is from the Basic Multilingual Plane and even the largest symbols have codepoints representable by 21-bits, a 4-byte codepoint is rarely needed. The codepoint value in UTF-32 is simply zero-padded to 32 bits [9].

UTF-32 suffers similar memory waste problems as wchar_t does, with the exception that its length is explicitly defined, and thus a data type representing an UTF-32 encoded character should always be able to encompass a 32-bit value range [7].

Similar to UTF-16, UTF-32 should either employ BOM, or one of its sub-encodings, UTF-32LE or UTF-32BE, to specify endianness (see 2.2.11).

Its less obvious advantage is that a UTF-32 codepoint can be used to retrieve a symbol from a Unicode table in constant time, whereas with UTF-8, an algorithm may need to look at up to 3 codepoints before and after the currently parsed codepoint to determine a symbol. UTF-16 also has to determine the symbol algorithmically since surrogate pairs may be used.

2.2.5 Universal Character Set

The Universal Character Set is an international standard and defines, among other, the UCS codespace — the character table. The UCS is applicable to the representation, transmission, interchange, processing, storage, input, and presentation of the written form of the languages of the world as well as of additional symbols [16].

UCS-2

UCS-2 is a big-endian two-octet fixed-length encoding able to represent the BMP. It was the encoding originally specified in the Unicode88 paper and known to be used by Windows versions older than Windows 2000. This encoding is now deprecated [16] in favor of its successor — UTF-16.

UCS-4

UCS-4 is another name for UTF-32, and the two terms can be used interchangeably [16].

2.2.6 Fullwidth and halfwidth characters

Some characters in the Unicode table are called fullwidth. In a nutshell, most latin characters occupy one character cell and are called halfwidth. However, they, as well as many characters from mostly Asian scripts have a so-called full-width representation, which occupies two character cells Distinguishing them is significant for correct output of many utilities. Figure 2.2 demonstrates the difference.

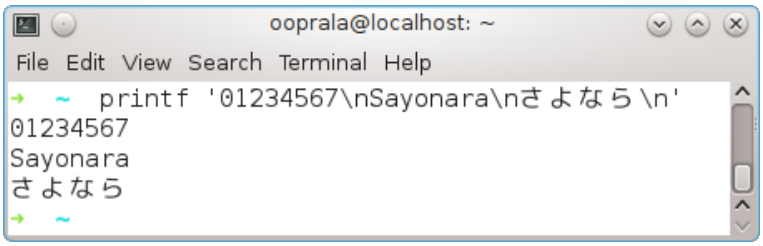


Figure 2.2: Halfwidth and Fullwidth characters

2.2.7 Planes and codepoints

A Unicode plane is a table of 256 rows and 256 columns. The current Unicode standard adopted the UCS — the Universal Character Set (see 2.2.5) and its division into planes. A plane is a table of characters consisting of 65536 symbols. There are 17 planes in total. The first of these planes is called the Basic Multilingual Plane, or BMP for short.

The remaining 16 planes are called supplementary planes, or sometimes astral planes, and their usage is very sparse [9]. They contain mostly ancient scripts, various symbols and special ideographs, used most often in names.

Basic Multilingual Plane

The BMP contains the most common modern characters, symbols, Japanese, Chinese and Korean ideographs. Apart from them, it also contains surrogate codes used in UTF-16 to represent characters outside of the BMP [7]. The surrogate mechanism is described in section 2.2.9. Figure 2.3 gives a concise idea of BMP's division.

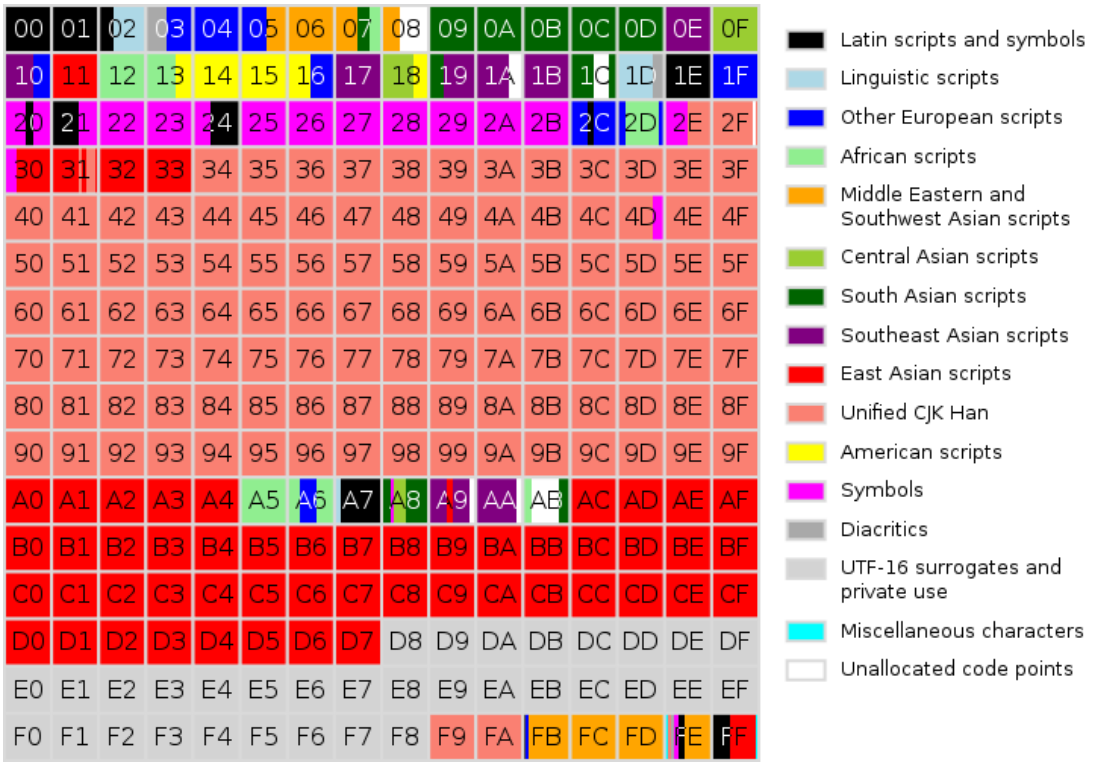


Figure 2.3: Division of the Basic Multilingual Plane[1]

2.2.8 Combining, decomposition and normalization of characters

This subsection briefly mentions character combining, explains the terms canonical decomposition, composite and normalization.

Combining

Many characters in the Unicode table can be represented in more than one way. This is often the case with the latin-1 characters. For example the letter 'é' has its own codepoint, and in this representation is called a *composite*. However, it can also appear *decomposed* as two separate Unicode characters: the character 'e' and the combining character '´'. This form is called a canonical decomposition of a character. By definition, every composite character has a canonical decomposition in Unicode. However, not every possible composition exists as a separate composite character in the table [9].

Sorting and equivalence

Decomposition imposes complexity especially on sorting. As an example, as far as Unicode is concerned, both "é" and "e´" are identical strings. In other words, both these representations are equivalent, and should have equal sorting weight. This imposes additional complexity on sorting, since sorting algorithms need to look ahead and backward to get complete characters. Since non-combining characters may have more combining characters attached to them, all characters have a numeric property called the combining class, ranging

[1] https://upload.wikimedia.org/wikipedia/commons/8/8e/Roadmap_to_Unicode_BMP.svg

from 0 to 255. All ”standalone” characters have a combining class of 0. Combining characters have a combining class from 1 to 255. A decomposed composite's combining characters are then sorted numerically using their combining classes, which is the only valid canonical decomposition of that character [9]. The basic rule is that a combining character sequence goes from one character with a combining class of 0 to the next character with a combining class of 0.

2.2.9 Surrogates

Surrogate pairs are a technique for UTF-16 encodings, used to represent characters outside of the BMP by using two UTF-16 words. They are the main difference between UTF-16 and UCS-2. A surrogate pair consists of a high surrogate (sometimes called a leading surrogate) and a low surrogate (sometimes called a trailing surrogate). Both of their legal values are ranges from the BMP and both have 1024 possible values. Surrogates should always appear in pairs, in other words, A leading surrogate can be followed only by a trailing surrogate and not by another leading surrogate, a non-surrogate, or the end of text [7], and an occurrence of one without the other implies that the string is ill-formed. Surrogates are rarely needed and an instance of their flawed implementation has been a source of a security vulnerability in the Python language parser in the past [23]. Figure 2.3 nicely shows the region reserved for surrogate codepoints in gray.

2.2.10 Compression schemes

Compression schemes appeared out of the most commonly voiced argument against the original Unicode proposal. That is, for US and Latin scripts in general, the size of all text data would double, which, at the time was a big problem for both storage and transmission. Later, with the standardization of UTF-8, many adopted it as a form of ”compression”.

SCSU

SCSU stands for Standard Compression Scheme for Unicode and was developed by Reuters. It is a file encoding scheme, not a compression algorithm. However, it uses compression algorithms itself, often yielding better results than if these algorithms were applied to non-SCSU Unicode-encoded text. SCSU is a stateful encoding, meaning a character may be represented differently in each occurrence, depending on the preceding characters. This implies, that SCSU-encoded text is not seekable and has to be interpreted sequentially from start to finish. It operates in two modes: single-byte mode and Unicode mode.

Single-byte mode In this mode, most ASCII characters, save several control ones, are representable as themselves. This implies, that a reasonable ASCII text is also a valid SCSU text. Latin-1 characters are also interpretable as themselves, so an ISO-8859-1 encoded text should be a valid SCSU text as well. Some ASCII control characters used for SCSU's special purposes are used to switch the interpretation of characters with codepoints 0x80 to 0xFF. The values always represent a contiguous block of codepoints from the UCS.

Unicode mode For scripts, where constant window switching would yield significant overhead, such as the Chinese Han script, there's the Unicode mode. SCSU's parser should switch to Unicode mode upon encountering a certain control character. All characters in this mode are considered as UTF-16BE encoded [9].

BOCU

BOCU stands for Binary Ordered Compression for Unicode. BOCU is a compression scheme developed by IBM and supersedes SCSU in matters of simplicity and compression [6]. Its greatest benefit is that BOCU-encoded text sorts in the same order as an uncompressed text would [9]. BOCU's downside is its per-request license. According to a letter from IBM, "IBM would like to offer a royalty free license to this patent upon request to implementers of a fully compliant version of BOCU-1" [5].

2.2.11 Endianness and Byte Order Mark

UTF-16 and UTF-32 are both byte-oriented and have their own ways of specifying the correct order of bytes after a transmission on a network.

Endianness Endianness refers to the computer architecture's approach to storing and reading memory. The smallest addressable unit being a byte, when manipulating a word, which is the CPU's address size (usually 32 or 64 bits), there's a choice to store the data from highest order byte to the lowest order byte or vice versa. This is a non-issue while on one machine, but becomes an issue once inter-computer communication is involved. The most common way for a Unicode text stream to denote its byte orientation is with the usage of a Byte Order Mark (see 2.2.11) at the beginning of text. Another way is specifying a Low-Endian (LE) or Big-Endian (BE) orientation in the encoding name (see 2.2.3 and 2.2.4).

Byte Order Mark A Byte Order Mark, often abbreviated as BOM, is an initial byte of a text stream, specifying unambiguously the endianness of the rest of the stream. It only makes sense for UTF-16 and UTF-32 text. A missing BOM implies Big Endian byte ordering. Using BOM is not necessary and actually discouraged in UTF-8. However, it often appears in UTF-8 encoded text anyway as a result of conversion from other encodings (UTF-16/UTF-32). Some developers use it to distinguish UTF-8 encoded text from other 8-bit encodings. One may come across the term UCS-4 [7], which is treated as a synonym to UTF-32, although it's defined in a different standard (see [16]).

Special meaning

The BOM was also used as "zero-width non-breaking space" until Unicode3.2. Having no visual representation itself, it was embedded between character codepoints that it was meant to glue together and inhibit breaking them into multiple lines. It has since been replaced with "WORD JOINER", a character dedicated to have the same functionality. An example of its usage would be to insert it between 'c' and 'h' in the Slovak language, where 'ch' is a separate character, thus ensuring the two letters are always shown together.

2.3 Unicode support in programming languages

The support for handling Unicode strings varies from language to language. For languages older than the standard, it's usually handled by either a third party library, an extension of the language standard (Python), sometimes both. Newer languages, such as Ruby, Perl, Go or Java, come with Unicode support from the very beginning. Some languages, such as PHP, have no standardized native Unicode support [22].

2.3.1 Java and .NET languages

Although both different technologies from different vendors, their internal representation of Unicode strings is almost identical. Both use UTF-16 internally and thus use UTF-16 surrogate pairs to represent characters outside the BMP [18] [17]. Both also have rich libraries supporting transparent character-oriented operations.

2.3.2 C and C++

As these languages are older than the Unicode standard, they did not initially have native support for multibyte characters [9]. Strings are instead represented as arrays of type char, which by the most recent standards, equals to one byte. One attempt at the solution to this issue were the wide characters (see 2.3.3). One can denote strings with wide characters with a leading 'L' before the string, making the compiler produce an array of type const wchar_t, instead of an array of const char.

```
char *string0  = "this is a string represented as an array of const chars";
wchar_t *string1 = L"this is a string rep. as an array of const wchar_t-s";
char *string2  = L"this will result in a compiler warning";
```

Another approach is that taken by the GNU libunistring library (see 2.3.5), where the array of chars is used to store UTF-8 strings, and the amount of bytes that constitute a character is determined algorithmically.

As of the C++11 standard, provides a way to natively represent Unicode strings [15], [4]. C++1z should further expand on these features and adds a similar way to represent UTF-8 character literals in the code [21]. The definition of byte in the C++ memory model was changed, so that "A byte is at least large enough to contain any member of the basic execution character set and the eight-bit code units of the Unicode UTF-8 encoding form and is composed of a contiguous sequence of bits, the number of which is implementation-defined" [15]. The standard also added literals u8, u, and U to denote UTF-8, UTF-16 and UTF-32 encoded strings, respectively. A typical example would be:

```
u8"I am an UTF-8 encoded string literal"
u"I am an UTF-16 encoded string literal"
U"I am an UTF-32 encoded string literal"
```

2.3.3 The Wide character type

The issue of not being able to properly represent and operate on strings with multibyte characters required mitigation even for lower-level languages such as C. Standardization attempts by the ISO C and POSIX committees resulted in a draft describing the wide character type and its respective API. The standard, however, is vague in certain points and several sections could be implemented solely based on the implementer's discretion.

Firstly, the wchar_t byte size was unspecified and is left up to the compiler writers' discretion. On Microsoft Windows and IBM AIX platforms, wchar_t is always 16 bits in length, which is insufficient to represent the entire Unicode table. On GNU/Linux systems, wchar_t is guaranteed to be 4 bytes in length. On BSD and Solaris, however, the wchar_t bytesize is not documented and depends on the locale chosen. On an architecture, where the ABI is defined in a way, where wchar_t is 4 bytes, storing a string consisting of ASCII-only characters would result in quadruple memory footprint of the string, almost making it

suitable for UTF-32. This is unfeasible not only for network communications, but for most applications in general.

Apart from that, the wchar-API does not have a 1-to-1 relation with the old C-string API. This was amended in a later POSIX iteration and an GNU glibc as an unofficial extension. The wide character standard is a failed effort, but often a necessary evil in many C/C++ applications up to this day. For performance reasons, among others, the current multibyte support for coreutils was never accepted by the upstream maintainers and only remains as a set of original li18nux movement patches kept in distribution packages (namely Fedora, RHEL and OpenSUSE).

2.3.4 International Components for Unicode

The International Components for Unicode, or ICU, is a set of mature, portable localization libraries for the C, C++ and Java languages developed by IBM. Apart from these languages, there exists 3rd party support for many more (Python and Ruby being only a few of the many) via wrappers. It is based on a set of Java internationalized libraries, and was originally written for Sun Microsystems. It contains conversion interface for both SCSU and BOCU compression schemes and is used by many open-source projects such as the Firefox and Chrome browsers, the Qt and boost libraries or the LibreOffice office suite, to name a few. It is licensed under its own ICU license compatible with GNU GPL [9].

2.3.5 Libunistring

Libunistring is a library of functions for manipulating text according to the Unicode standard. It is very well-tested, as many of its modules originally come from gnulib. In the past two years, the libunistring mailing list consisted mainly of conversations about build failures on exotic architectures or enhancing support for the new standard, bug reports were very rare. Originally written by Bruno Haible, the author of the clisp interpreter (among other), the library has been accepted by the GNU project as the main tool for GNU applications that work with strings [3]. It is often distributed as a stand-alone library. However, it's also a part of the GNU gnulib portability library[2] and shares its design goals, making it an extremely portable solution to this problem. It provides full support for the Unicode 7.0.0 Standard.

Libunistring is currently not very wide-spread, and is used mostly by lesser-known open-source projects such as Tracker(database), Hop (Web 2.0 programming language), Rygel (multimedia sharing) or Guile (GNU implementation of Scheme). However, it's also a dependency of GNU gettext, a utility aimed at providing multilingual documentation and messages in programs. Updated information on libunistring's usage can usually be found by querying the package system on most Linux distributions.

Libunistring was written mainly as a smaller, more flexible Unicode handling library, that would implement only the necessary parts of ICU. Another motivation was also that it would be a GNU project, making it easier to push through bugfixes and changes, than it would be for a robust library from a different vendor.

2.3.6 Python — a Case Study

As a demonstration of how difficult it can be to assess the drawbacks and benefits of using any of the standard encodings during design, one can look at the Python language's Unicode

[2] https://www.gnu.org/software/gnulib/MODULES.html#module=libunistring

type internal representation evolution over time [19]. This also demonstrates how difficult the problem of reliably representing text is.

The first implementation of the *unicode* class in Python2.0 used UTF-16 internally, and without the use of surrogates, thus only supporting the Basic Multilingual Plane. The proposal hinted a possible 32-bit extension of the language in the future [11]. The extension came with a Python Enhancement Proposition a year later in PEP261 [20]. It extended the Python internals to support internal string representation using UTF-16 with the support of surrogate pairs or UTF-32 switchable at compile time. With Python3.0, Unicode type became the default string type of the language [19]. The official internationalization of the Python3 language brought forward several issues for POSIX systems, converting certain data to Unicode and creating surrogates where there were none [12]. With Python3.3, the internal representation changed, so the encoding was set by the character with the highest ordinal. For example, UTF-8 for ASCII-only text, UTF-16 for text containing common Kanji script and UTF-32 for strings containing ancient scripts such as Gothic [13].

2.4 The Linux Internationalization Movement

The Linux Internationalization Movement, often abbreviated Li18nux and later renamed to OpenI18N, stands for the Linux Internationalization Initiative. The ”i18n” stands for internationalization, the number 18 signifying that there are 18 letters between 'i' and 'n'. It is a specification for the internationalization capabilities that should be included in a Linux implementation. Among others, it mandates the use of ICU (see 2.3.4). It is now a module of the much larger Linux Standard Base Specification (LSB [24]), which was created to lower the overall costs of supporting the Linux platform.

Chapter 3

Design of the multi-byte support

The GNU Core Utilities[1] are the basic file, shell and text manipulation utilities for the GNU/Linux distributions and many flavors of Unix. These are the core utilities which are expected to exist on every operating system . This chapter briefly describes the process of adding multi-byte support to the coreutils package, first attempts as well as the currently accepted solution, it mentions the Li18nux movement efforts regarding coreutils and the movement's pitfalls and also showcases the upstream-proposed solution that is already under construction. This is the effort this thesis is attempting to contribute to.

3.1 Initial status

Coreutils, dating back to as far as 1992, was formed by three previously stand-alone packages — fileutils, shellutils and textutils, neither of which had any locale-awareness. Originally, the coreutils package utilities were completely unaware of the locale they were running in, mostly only considering ASCII-encoded input and working with the assumption that one byte equals one character. As the UTF-8 adoption trend graph 2.1, this is not an ideal approach for several years already. The original multi-byte patches, which are still used by a few distributions at the time of writing, originate from the year 2000 and the Li18nux movement. The patches were never accepted by the upstream project maintainers due to its mechanical approach, code duplication, several security issues and a negative performance impact [3]. The coreutils project maintainers have assessed the pros and cons of current solutions and deemed the original Li18nux movement patches unsuitable for upstream inclusion. However, several distributions still maintain these patches and include them in their coreutils packages.

3.2 Proposed solution

Several years after the Li18nux movement, Bruno Haible released the Libunistring library (see 2.3.5), which, for coreutils, is to be used as a final solution to the multi-byte issue. Pádraig Brady, the current upstream maintainer put together a plan to add multi-byte support to the coreutils set of utilities, specifying several goals. The main goal is the re-implementation of multi-byte support for the utilities covered by the original Li18nux patches, so that they may be dropped from distributions [3]. Furthermore, the utilities' op-

[1] https://www.gnu.org/software/coreutils/

eration in C locales should not be adversely affected by the multi-byte extensions, regarding both performance and correctness.

I've chosen six utilities to convert — expand, unexpand, cut, fold, fmt and paste. Their algorithms and usage are very different and are further described in the following pages. The choice was made based on several factors. Firstly, the aforementioned *coreutilsi*18*nplan* has its own preferred order or internationalizing utilities, which I mostly adhered to. Also, several of these utilities had work already done on them by other contributors. Their patches either wait for review or they expressed their commitment to further work.

3.3 expand and unexpand

The expand and unexpand form a tight couple, one fundamentally do the inverse of the other. However, their algorithms differ.

In a nutshell, expand converts tabs in given files to a certain number of spaces and outputs them to standard output, maintaining the proper alignment at tab stops. Backspaces in the input are preserved, but they also decrement the tab counter. Additionally, users may specify the positions of their own tabstops, making the expansion of input with fullwidth characters non-trivial. Unexpand, conversely, transforms a specified number of consecutive spaces at desired locations into tabulators. By default, it only converts blank characters at the beginning of each line. Backspaces are again preserved.

With expand and unexpand, it turned out to be easy to factor out common parts, thus there is a third module added called `expand-core.c`, with common functions of both expand and unexpand. Like other utilities, it considers one byte to be one character. The code was amended to recognize multibyte characters and especially characters in their "fullwidth" forms, occupying more than one character cell (see 2.2.6). The algorithm thus had to be changed to account for the number of characters as well as the number of character cells they occupy. The algorithms were changed to adhere to those requirements.

```
initialization;
foreach character do
    if character == tab then
        repeat
            increment column;
            print space;
        until column == next_tabstop;
        continue;
    end
    else if character == newline then
        column = 0;
    end
    print character;
    column += character_width;
end
```

Algorithm 1: Algorithm of Expand

```
initialization;
foreach character do
    if character == space then
        increment consecutive_spaces;
        continue;
    end
    else if character == newline then
        column = consecutive_spaces = 0;
        continue;
    end
    else if column + consecutive_spaces == next_tab_stop then
        print tab;
    end
    else
        while consecutive_spaces > 0 do
            print space;
            decrement consecutive_spaces;
        end
    end
    print character;
    increment column;
end
```

Algorithm 2: Algorithm of unexpand when all blanks are converted (unexpand –all)

3.4 cut

Cut is a utility which is used to divide lines by bytes, characters or fields, where fields are delimited either by a tab, or by a user-specified character. All ranges are specified in the form of `N-M`, where N denotes the starting and M denotes the ending byte, character or field. While division by bytes is relatively straightforward and needs no change, the dividing by characters and diving by fields features need to be Unicode-aware. There is also a mode where cut divides by bytes, but preserves valid multibyte characters together. The user can also specify if non-delimited lines are to be printed and even specify an output delimiter when in byte or character mode.

All division may be specified by ranges. Here, again, is the problem of distinguishing characters and comparing them to the delimiter properly. Algorithm 3 illustrates cut's internals in more detail.

3.5 fmt

The fmt utility, as its name implies, formats text to a desired width without splitting words, and reformats indentation, optionally detecting prefixes. Here again, byte-oriented text processing took place and the fmt utility was patched to properly recognize word widths and thus properly format the output lines. By default, fmt also expands tab characters and preserves additional spaces between words or lines. Fmt further uses a few heuristics to determine when to break sentences for optimal output. For example, if it is possible, it doesn't split lines after the first word and before the last word of a sentence. Lastly, fmt has

```
initialization;
foreach line do
    while true do
        find delimiter in line;
        if not delimiter_found then
            if print_nondelimited_lines then
                print line;
            end
            break;
        end
        else if field_number in wanted_fields then
            print line[current_position, delimiter_position];
            print output_delimiter;
        end
        current_position = delimiter_position + 1;
    end
end
```

Algorithm 3: Algorithm of Cut when separating fields

several modes of operation, where it only formats lines beginning with a certain prefix, or doesn't format the beginning lines of a paragraph, can squash redundant spacing between words or work in a split-only manner, where shorter lines are not joined together, which is often desirable for formatting code. Algorithm 4 illustrates fmt's internals in more detail.

```
initialization;
foreach paragraph in text do
    process words in paragraph;
    format paragraph based on specified width;
    output paragraph;
end
```

Algorithm 4: Fmt's algorithm with default options

3.6 fold

The fold utility simply rounds the lines to the specified width or character count, concatenating the rest with the following line. Characters and especially their widths have to be accounted for. Additional complexity comes with the option to break at spaces, where simple character counting is not enough. Algorithm 5 illustrates fold's internals in more detail.

3.7 paste

The paste is a simple utility that merges corresponding lines of the files given as arguments. The issue here is that the user may specify a list of delimiter characters, which may of course be any valid Unicode characters, with varying column and byte width. This list of delimiters is then cycled through and continuously inserted between the merged lines.

```
initialization;
foreach line do
    foreach char in line do
        if column + char_width > folding_column then
            print newline;
            column = 0;
        end
        print character;
    end
end
```

Algorithm 5: Fold's character folding algorithm

Three main things were changed in this utility. Firstly, the function to filter escape characters from the delimiter string uses `u8_next` to iterate over the string, as it needs to be multibyte-aware. As we do not have a Unicode equivalent of `getchar()`, the utility buffers entire newline-delimited lines instead. This is not wasteful, as the utility uses `getc()` in a loop anyway. Finally, instead of an array of delimiting characters, the list of delimiters is iterated over using a `u8_next()` function, making it character-aware. Algorithm 6 illustrates paste's internals in more detail.

```
initialization;
while true do
    for file in input_files do
        line = read_line from file;
        print line;
        print next_delimiter;
        if end_of_file then
            close file;
            if all_files_closed then
                end program;
            end
        end
    end
    print newline;
end
```

Algorithm 6: Algorithm of the paste utility

Chapter 4

Implementation of multi-byte support in the GNU coreutils

In this chapter, I describe in greater detail the implementation and its related problems, talk extensively about testing, which is an integral part of the process and assess the performance impact of the Li18nux and libunistring approaches to respective utilities. Unless otherwise noted, all examples are written in the C language, except when changing the configuration and building scripts, which are handled by autotools. This chapter contains a more detailed description of the implementation approach I've taken. It also showcases several important testcases, describes the correct output, shows original incorrect output and mentions coreutils tests in general.

Next, section 4.3 compares performance with ASCII input in the classic C locale using the original code base, then with multi-byte inputs processed by utilities patched with the Li18nux patches, which use the standard C functions for wide character and string handling, and inputs processed by utilities using libunistring.

4.1 Common code sections

The programs bundled in the coreutils package share a few common traits. For one, they share a common build system. They are also linked with the libcoreutils library, which is mostly a bundle of required modules from the Gnulib library. Several lines of code are common to all programs, such as added linking options in the `src/local.mk` script or proper `#include` directives. Control sequences to iterate through strings have also changed and are basically identical for all programs. Apart from these code segments, other portions of the code are mostly specific to the application and do not lend themselves to reusability. All the utilities are single-threaded. The source code has grown by 27% in linecount on average. Table 4.1 provides more detail.

4.1.1 while loops

Since we can no longer assume that one `char` equals one `byte`, we have to use a function which can iterate over characters.

Utility	lines(orig.)	lines(li18nux)	lines(thesis)
paste	522	N/A	719
cut	831	1242	1169
expand	430	593	339[1]
unexpand	532	758	482[1]
fmt	1033	N/A	1196
fold	308	571	544
Average growth (where applicable)		51%	27%

Table 4.1: Code growth for each utility (measured by wc -l)

Listing 4.1: Classic C-style string processing.

```
char *string = input_string;
char c;
while ((c = *string++) != '\0')
  {
       /* Process character c */
  }
```

The classic char* loop above needed to be rewritten to either a vector of UTF-32 characters represented by `ucs4_t` or to a UTF-8 string.

Listing 4.2: UTF-8 string processing.

```
uint8_t *string = input_string;
ucs4_t c;
while ((string = u8_next (string, &c)) != NULL)
  {
         /* Process Unicode character c */
  }
```

4.1.2 The development environment

As for the implementation environment, all the programs are written in GNU C99, although no GNU extension to the language are actually used. The compilers being used are the GCC-4.9.2 on Fedora 21 and a manually built GCC-5.1. However, compiling the source code with Clang, and on other Linux or Unix flavors should pose no problems, assuming all dependencies are present on the system. The source code was written on top of Coreutils v8.23, bootstrapped with Libunistring 0.9.3, using GCC 4.9.2 and GCC 5.1. The resulting binaries have grown approximately by 2% compared to the binaries compiled with code patched with the original Unicode support.

Bootstrapping libunistring is a process with several steps. Since libunistring is a gnulib module, it is added to the list of `gnulib_modules` in the file `bootstrap.conf`. Furthermore, the `gl_LIBUNISTRING` m4 macro needs to be placed in the `configure.ac` file. The `gl_LIBUNISTRING` macro ensures the proper definitions such as

```
HAVE_LIBUNISTRING=yes
```

[1] Code refactored to expand-core.c is additional 188 lines long

if libunistring is found. Among others, it also defines the `LIBUNISTRING` macro that expands to necessary linking information.

The coreutils build system needs to be aware of the specific utilities' need for linking with libunistring. For cut, for example, this would be done by adding a following line to the Automake `src/local.mk` file, written in autotools syntax.

```
src_cut_LDADD += $(LIBUNISTRING)
```

4.1.3 Common code

Each internationalized utility contains this definition:

```
static bool u8_locale;
```

Being a static variable, it's zero-initialized by default. For utilities compiled with libunistring support, `u8_locale` can be set to true if certain conditions hold.

```
#if HAVE_LIBUNISTRING
u8_locale = STREQ ("UTF-8", locale_charset());
#endif
```

The `u8_locale` locale variable is used to determine whether converting input text to UTF-8 is really necessary.

4.1.4 Outputting separate characters

It is often desirable to output a string character by character. The common function for this would be `putchar()`. To output separate Unicode characters, the libiconv library provides the `print_unicode_char()` function, which is declared in the `unicodeio.h` header file.

4.2 Testing the implementation

The testing was done on the following machine:

CPU Intel(R) Core(TM) i5-2540M @ 2.60GHz with two cores, four threads and a 3072 KiB L3 cache

RAM 8GiB

GPU Intel HD Graphics 3000 GPU

HDD 320GB ATA HDD Toshiba with 7200 RPM

System GNU/Linux Fedora 21

The binaries were compiled with GCC-5.1 with options "-O2 -g". The functionality testing is a combination of white-box and black-box testing, as tests are often based on the knowledge of source code itself, not just the described functionality.

Tests have been taken from three sources. There are multibyte tests used in Fedora, which are not in upstream repositories. Then there are patches I've written myself, based on the knowledge of the internals. The last source is made of translated output strings of

common programs taken from the Fedora Zanata[2] localization project. Most of these tests are written in either perl or shell script.

If you're going to test the original, unpatched programs to see for yourself, I strongly encourage you to pipe the output through 'cat -A' to see the non-printing characters, and special characters with no visual representation. If you want to run the complete coreutils testsuite, you can do so with the "make check" command. Tests can be found either in the `tests/<utility>/` directory or sometimes separately in a `tests/misc/<utility>.pl` script. The tests I've written were written mostly to fill the gaps left by the Fedora tests, or where tests were missing altogether, since the Fedora multibyte test coverage is very large.

Thorough testing is necessary for any new feature, especially for solutions like multibyte support, that often require larger changes. This section shows a deficiency with each converted utility on a simple test. More elaborate testing can be achieved with larger inputs, such as those used for performance testing (see 4.3).

4.2.1 expand and unexpand

As mentioned before, expand and unexpand are used to convert between tabs and spaces in text, mostly to preserve indentation. Having the following input:

```
1 ä	ö	ü	ß
```

a Unicode-aware version of expand seems virtually unchanged, if one does not closely examine the whitespace:

```
a       b       c       d
ä       ö       ü       ß
```

Conversely, the original expand would output:

```
a       b       c       d
ä      ö      ü      ß
```

Expand would originally garble the output, considering each of the characters on the second line as being two single-byte characters containing two cells total. The new code doesn't consider the byte length of a character but rather its width when printed.

4.2.2 cut

Due to the cut utility's way of transforming input, multi-byte input is garbled much more severely than in the case of expand and unexpand, where human-readable text would still be legible, even after improper expansion/unexpansion. The cut utility offers several modes to work with input. There is division by individual bytes, which is left untouched. However, a list may be specified to only cut out certain characters, fields or their ranges, and this portion of the code has to be multi-byte aware. The original multi-byte support also implements a mode where bytes generally get split but multibyte characters do not. Here is an example of running the original `cut --characters=2-4`, where we only want the second, third and fourth character from every input line, Having the input line:

```
1 äöüßü
```

[2] https://fedora.zanata.org/

a Unicode-aware version of cut outputs:

```
öüß
```

In the classic upstream version of cut, the output consists of the second byte of 'ä', the 'ö' letter and the first byte of 'ß'.

4.2.3 fmt

Fmt's deficiencies can be made pretty obvious with specifying a smaller width. Let there be a file IN containing

```
1 častica X
```

which means "particle X" in Slovak.

The patched fmt correctly counts characters and doesn't split a line 9 characters long, outputting the IN file unchanged:

```
častica X
```

The old fmt would count bytes instead of characters and when invoked as `fmt --goal=9 ./IN` would output:

```
častica
X
```

Fmt has two basic modes — counting characters and counting width, so both have to be accounted for when processing the input.

4.2.4 fold

As mentioned before, fold compacts the input lines to a certain number of characters per line, prepending the remaining characters to the next line. However, with fullwidth or multibyte characters, output gets malformed quickly. Trying `fold --width=2` on the following input:

```
1 äöüßü
```

should give this output:

```
äö
üß
ü
```

Instead, the original utility counts bytes, considering each of these characters as being two cells long, resulting in the following output:

```
ä
ö
ü
ß
ü
```

4.2.5 paste

Paste being one of the utilities without any multibyte support whatsoever, a deficiency can be found very easily. Let there be a file `file1`, with the following contents:

```
1
2
3
```

and with `paste` invoked in the following manner:

```
paste file1 file1 -d 'æ'
```

The output of a Unicode-aware paste utility is:

```
1æ1
2æ2
3æ3
```

Instead, the original output of `paste` is simply the contents of the file1 with a few non-printing trailing characters. With the proper patch, *paste* now handles multibyte characters of any width correctly.

4.2.6 Results

This section illustrated basic deficiencies in the original utilites, when it comes to Unicode handling. The converted utilites have been run through the original upstream tests, multibyte tests from the Fedora coreutils repository and also tests written by myself with success.

4.3 Performance evaluation

This section showcases the performance impact of the original Li18nux approach, compared to my own, where applicable. The results were measured by the *time* utility. The environment used for performance testing is the same as the one used for functionality testing. For testing UTF-8, the locale en_US.UTF-8 was used. For 8-bit locales, the standard "C" locale was set.

4.3.1 Sources of performance test data

Performance was measured with an input of almost 2 million lines of Japanese text. More precisely, input was formed of a concatenation of the book American Stories by Kafu Nagai a 1000 times over. The resulting file is 448MiB big and 171,989,000 characters long. The file is further referred to as LARGE.txt. This approach was selected over running the utility a 1000 times with the original book, since the measurements would include the overhead of creating 1000 processes. The book was downloaded from project Gutenberg[3]. This provides a very strong test input, as the book contains a mixture of US ASCII text in the preface with fullwidth Japanese ideographs with a byte length of either 2 or 3 mixed with poems in French, containing many French-specific letters. For the testing of 8-bit

[3] https://www.gutenberg.org/

encodings, an ASCII version of Alice in Wonderland by Lewis Carroll is used. The book was converted from UTF-8 from project Gutenberg and concatenated into a file 100,000 times over, resulting in a file 72MiB large. It shall be refered to as ALICE.txt in the following tables. The books can also be found on the accompanying CD, in the directory *perf*. For each utility, the most common usage was selected, based on the info or manual pages, sometimes modified to be more Unicode-centric. To keep the examples as legible as possible, multibyte characters used in arguments are printed via escape sequences.

4.3.2 expand and unexpand

Most literature doesn't contain enough tabs or consecutive spaces to make itself an interesting testcase for expand and unexpand. To compensate for that, I've substituted every ideographic full stop in the `LARGE.txt` file with a tab. The file shall be further referred to as `LARGE_EXPAND.txt`. The resulting output of expand is used as a test to `unexpand --all`, and shall be called `LARGE_UNEXPAND.txt`. For testing 8-bit encodings, I've similarly substituted full stops with tabs in the file `ALICE.txt`. The files used for testing shall be refered to as `ALICE_EXPAND.txt` and `ALICE_UNEXPAND.txt`.

utility with parameters	li18nux[s]	thesis[s]	difference
expand LARGE_EXPAND.txt	7.76	7.35	5.43%
unexpand LARGE_UNEXPAND.txt	8.71	8.46	2.91%

Table 4.2: Performance comparison of expand and unexpand - UTF-8 locale

utility with parameters	li18nux[s]	thesis[s]	difference
expand ALICE_EXPAND.txt	0.34	0.53	43.67%
unexpand ALICE_UNEXPAND.txt	0.93	1.10	16.75%

Table 4.3: Performance comparison of expand and unexpand - C locale

Surprisingly, expand and unexpand scored the smallest performance increase of the converted utilities. It is also the only one which slowed down a little when handling 8-bit locales.

4.3.3 cut

For cut, I've tried cutting out a range of characters, cutting fields separated by an ideographic full stop, and cutting a complement of 4 selected characters. The `-c`, or `--characters` option was not present in the original code.

cut parameters	li18nux[s]	thesis[s]	difference
-c2-32 LARGE.txt	5.23	2.78	61.17%
-f1,3 -d LARGE.txt $(printf \U3002)	5.41	2.93	59.47%
-c1,3,5,7 –complement LARGE.txt	5.76	3.29	54.59%

Table 4.4: Performance comparison of cut run in UTF-8 locale

cut parameters	li18nux[s]	thesis[s]	difference
-b2-32 ALICE.txt	0.21	0.19	10%
-f1,3 -d '.' ALICE.txt	0.24	0.22	8.70%

Table 4.5: Performance comparison of cut run in C locale

4.3.4 fold

I've tried the two modes that have changed during conversion of fold. One was to fold to lines 40 characters long. The second was to fold to the specified width of 60 columns. The `-c`, or `--characters` option was not present in the original code.

fold parameters	li18nux[s]	thesis[s]	difference
-c40 LARGE.txt	6.59	3.64	57.67%
-w60 LARGE.txt	7.40	4.39	51.06%

Table 4.6: Performance comparison of fold in UTF locale

fold parameters	li18nux[s]	thesis[s]	difference
-b60 ALICE.txt	0.41	0.38	7.59%

Table 4.7: Performance comparison of fold in C locale

4.3.5 paste and fmt

Paste and fmt do not have any prior Unicode support to interfere with C locales, thus I'm only comparing to the bare original, ASCII-only code.

utility	upstream[s]	thesis[s]	difference
paste -s ALICE.txt	0.26	0.29	10.9%
fmt	2.45	2.72	10.44%

Table 4.8: Performance comparison of paste and fmt

4.4 Results

The performance testing clearly shows speed up in executing, when operating under a Unicode locale. It also demonstrates that operation under a classic "C" locale wasn't impacted in most utilites, apart from a slight slowdown in expand and unexpand.

Chapter 5

Conclusion

The aim of this thesis was to assess the Unicode support of one of the most basic set of utilities present on the GNU/Linux systems and either augment or add it if it's either unsatisfactory or non-existent. The thesis investigated the extent and quality of the current multibyte support in the GNU Core Utilities, described the history, motivation and the current status, followed by an attempt at robust implementation of the Unicode support. It also briefly summarized the Unicode standard. All researched literature and other sources are duly referenced.

My solutions was benchmarked and compared to the existing solution. I've also described the new upstream approach in-depth and used it as a guideline in my implementation, focusing mainly on the libunistring module, but also on the larger gnulib library and its independent multibyte modules. The new features were written in such a way as not to impact performance for single-byte locales, where the utilities already perform properly. Most programs were written to internally use UTF-8, being very fast for UTF-8 encoded input consisting of ASCII-only characters and thus not impacting performance at all or only very minimally.

The changes were implemented in 6 programs, namely cut, expand, fmt, fold, paste and unexpand. The programs were properly tested and benchmarked against the li18nux movement's solution. The resulting patches were also sent to upstream maintainers for further comment. The patches add approximately 1000 lines of new code plus a few lines of configuration, and around 30 tests. For most utilities, the number of tests has more than doubled, compared to upstream. The average source code growth in linecount for these utilities was only 27% as opposed to 51% for the Li18nux solution, with the smallest proportional growth needed for expand and unexpand due to refactoring and the largest growth for the utility fold. The speedup in handling Unicode input was up to 61%.

However, the complete localization of the coreutils package is still far from done. There are more utilities which require patching, some very difficult to tinker with, such as sort or uniq, where the upstream developers have yet to decide where to draw the line. Some utilities work with regular expressions, which the libunistring library doesn't provide an API for yet. Moreover, upstream inclusion of larger features often takes years.

I'm considering several next steps for future work. Since the libunistring library itself is not yet complete, implementing the remaining support would prove very useful. There are several alternatives to the standard single-byte string handling functions that are missing from the library and the regular expression module is missing completely. Regarding coreutils, upstream should be given more opinions and requests from users and contributors to be more inclined to deal with problems not yet solved in the previously mentioned up-

stream plan. Next, there are more difficult utilities such as sort, join or uniq, which share a common code base and are very sensitive to code change. These utilities need more time allocated to their conversion.

Bibliography

[1] Joseph D. Becker. Unicode88. Technical report, Unicode Consortium, California, 1998. URL http://unicode.org/history/unicode88.pdf.

[2] R. W. Bemer. American Standard Code for Information Interchange. Technical report, American Standard Association Incorporated, New York, 1963. URL http://worldpowersystems.com/J/codes/X3.4-1963/.

[3] Pádraig Brady. A plan for coreutils i18n support. online. URL http://bugs.python.org/issue14579.

[4] Lawrence Crowl and Beman Dawes. Raw and Unicode String Literals, Unified Proposal (Rev. 2). Technical report, ISO/IEC, Geneva, 2007. URL http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2442.htm.

[5] Mark Davis and Markus W. Scherer. UTN #6: BOCU-1. online. URL http://www.unicode.org/notes/tn6/.

[6] Mark Davis and Markus W. Scherer. Binary-Ordered Compression for Unicode. Technical report, IBM, Dallas, 2001. URL https://ssl.icu-project.org/docs/papers/binary_ordered_compression_for_unicode.html.

[7] Julie D. Allen et al. *The Unicode Standard, Version 7.0*. The Unicode Consortium, Mountain View, California, 2014. ISBN 978–1–936213–09–2.

[8] Eric Fischer. The Evolution of Character Codes, 1874-1968. Technical report, Universitá Degli Studi Di Trieste, Trieste, Italy, 2002. URL https://web.archive.org/web/20050305043226/http://www.transbay.net/~enf/ascii/ascii.pdf.

[9] Richard T. Gillam. *Unicode Demystified*. Addison-Wesley Professional, 1 edition, 2002. ISBN 978–0-201700–52–7.

[10] Jukka K. Korpella. *Unicode Explained*. O'Reilly Media, 1 edition, 2006. ISBN 978–0-596101-21–3.

[11] Marc-André Lemburc. Python Unicode Integration. Technical report, Python Software Foundation, Wolfeboro, New Hampshire, 2000. URL https://www.python.org/dev/peps/pep-0100/.

[12] Martin v. Löwis. Non-decodable bytes in System Character Interfaces. Technical report, Python Software Foundation, Wolfeboro, New Hampshire, 2009. URL https://www.python.org/dev/peps/pep-0383/.

[13] Martin v. Löwis. Flexible String Representation. Technical report, Python Software Foundation, Wolfeboro, New Hampshire, 2010. URL https://www.python.org/dev/peps/pep-0393/.

[14] WWW page. Unicode 8.0.0. online. URL http://unicode.org/versions/Unicode8.0.0/.

[15] WWW page. Working Draft, Standard for Programming Language C++. Technical report, ISO/IEC, Geneva, 2013. URL http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf.

[16] WWW page. Information Technology - Universal Coded Character Set(UCS). Technical report, ISO/IEC, Switzerland, 2014. URL http://standards.iso.org/ittf/PubliclyAvailableStandards/c063182_ISO_IEC_10646_2014.zip.

[17] WWW page. String (Java Platform SE 7). Technical report, Oracle Corporation, California, 2014. URL http://docs.oracle.com/javase/7/docs/api/java/lang/String.html.

[18] WWW page. Unicode in the .NET Framework. Technical report, Microsoft Corporation, Washington, 2015. URL https://msdn.microsoft.com/en-us/library/9b1s4yhz%28v=vs.90%29.aspx.

[19] Benjamin Peterson. The Guts of Unicode in Python. online. URL http://pyvideo.org/video/1768/the-guts-of-unicode-in-python.

[20] Paul Prescod. Support for 'wide' Unicode characters. Technical report, Python Software Foundation, Wolfeboro, New Hampshire, 2001. URL https://www.python.org/dev/peps/pep-0261/.

[21] Richard Smith. Adding u8 character literals. Technical report, ISO/IEC, Geneva, 2014. URL http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4267.html.

[22] Victor Stinner. Programming with Unicode. Technical report, France, 2010. URL https://unicodebook.readthedocs.org/en/latest/index.html.

[23] Serhiy Storchaka. CVE-2012-2135: Vulnerability in the utf-16 decoder after error handling. online. URL http://bugs.python.org/issue14579.

[24] Linux Standard Base team. The Linux Standard Base: Reducing Complexity for ISVs Targeting Linux. 2008. URL https://www.linux.com/learn/whitepapers/doc/5/raw.

[25] David A. Wheeler. Secure Programming for Linux and Unix HOWTO. online. URL http://www.dwheeler.com/secure-class/Secure-Programs-HOWTO/character-encoding.html.

Appendix A

CD Contents

The CD contains the following files and directories:

coreutils Directory containing the base code of coreutils-8.23 with fedora tests, patched with my changes. The code can by compiled by running `./configure` and `make`. The tests can be run by `make check`.

patches Directory containing my patches separately.

patches/i18n.patch The original Li18nux patchset.