GNU libc

New Features in GNU libc 2.2

Ulrich Drepper
Cygnus Solutions, a Red Hat Company
`drepper@redhat.com`

History

`glibc 2.0.*`

- □ released in January 1997
- □ functionality of ISO C99 (as of this time)
- □ contained 8-bit character set support without the extended

`glibc 2.1.*`

- □ released in February 1999
- □ contained all missing ISO C99 support except for wide-character streams
- □ Unix function **`iconv()`** implemented with 120+ character sets supported
- □ working **`mb*towc*()`** and **`wc*tomb*()`** functions

But: still no multibyte character locales supported!

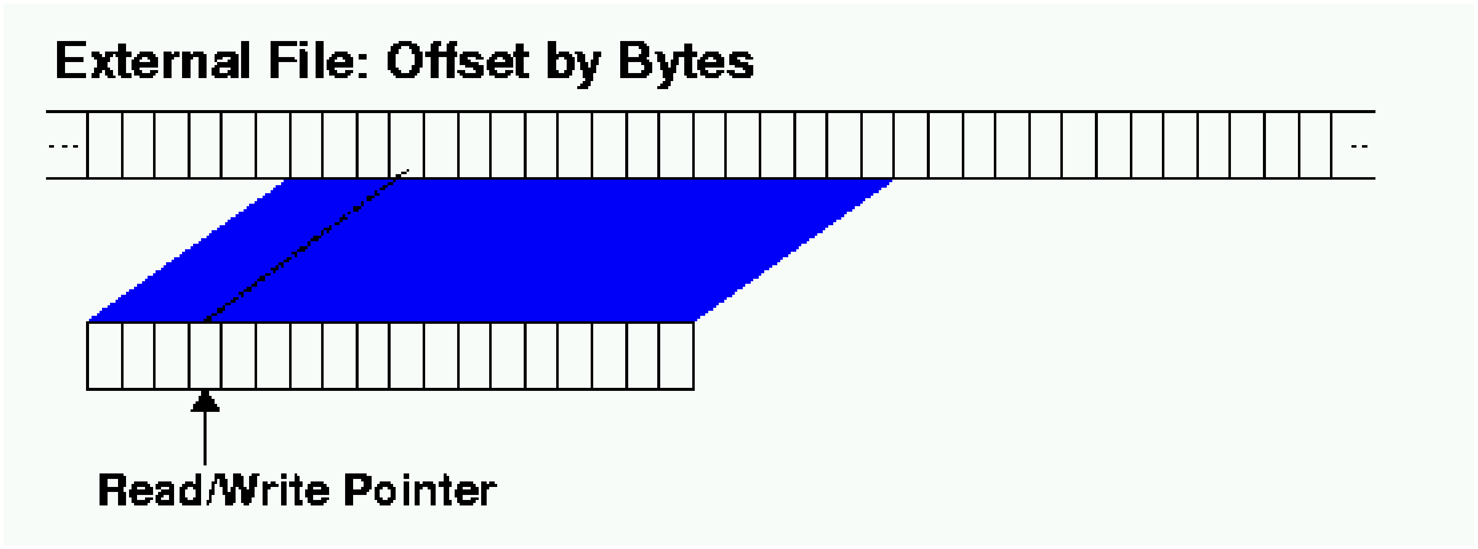
Future

`glibc 2.2.*`

- □ released 200x (don’t ask when!!!)
- □ complete internationalization support
  - ○ wide character streams
  - ○ multibyte locales
  - ○ ISO 14651 collation
  - ○ missing support in, e.g., **`fnmatch()`** and **`regex()`** added
- □ extended internationalization features
  - ○ on-the-fly conversion of message catalogs
  - ○ handling of plural forms in message translations
  - ○ transcription and transliteration for conversion to unsupported scripts and character sets
- □ up-to-date IPv6 library support

Wide Character Streams, I

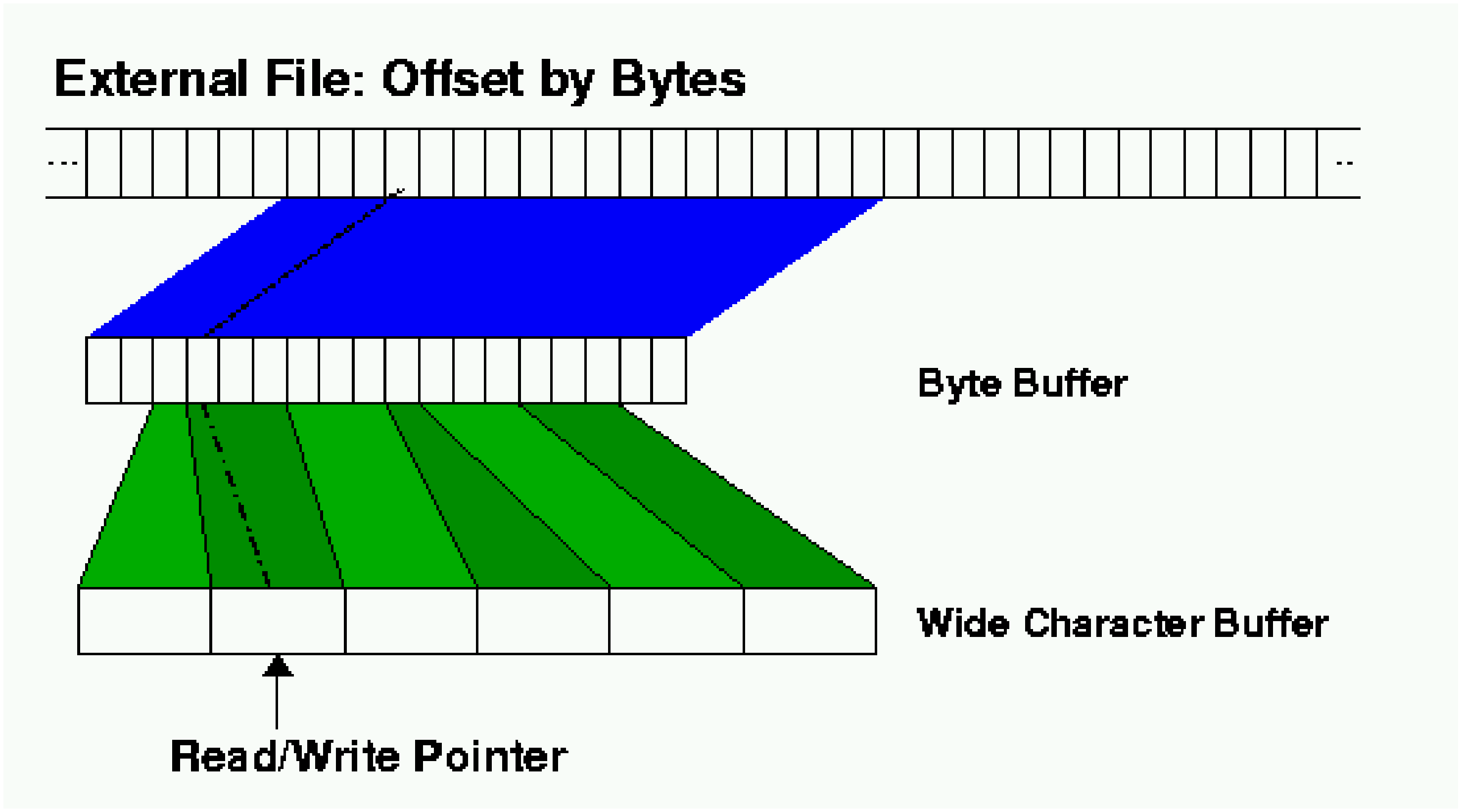
Normal Streams:



- ☐ 1:1 relation between bytes in the file and in the buffer
- ☐ no special support for multibyte characters (at least not officially)
- ☐ direct correspondence between read/write pointer and file position
- ☐ fast!

Wide Character Streams, II

Wide Character Streams:



- two buffers used
- complex relation between
  - the two buffers
  - the three read/write pointers

Multibyte Locales

`localedef` now can handle multibyte character sets:

- ☐ wide-character collation is implemented correctly
- ☐ character width information available through **`wcswidth()`**
- ☐ character transliteration information from ISO 14652 used
- ☐ era time format handling corrected fo Japanese locales

Also:

- ☐ new categories following ISO 14652

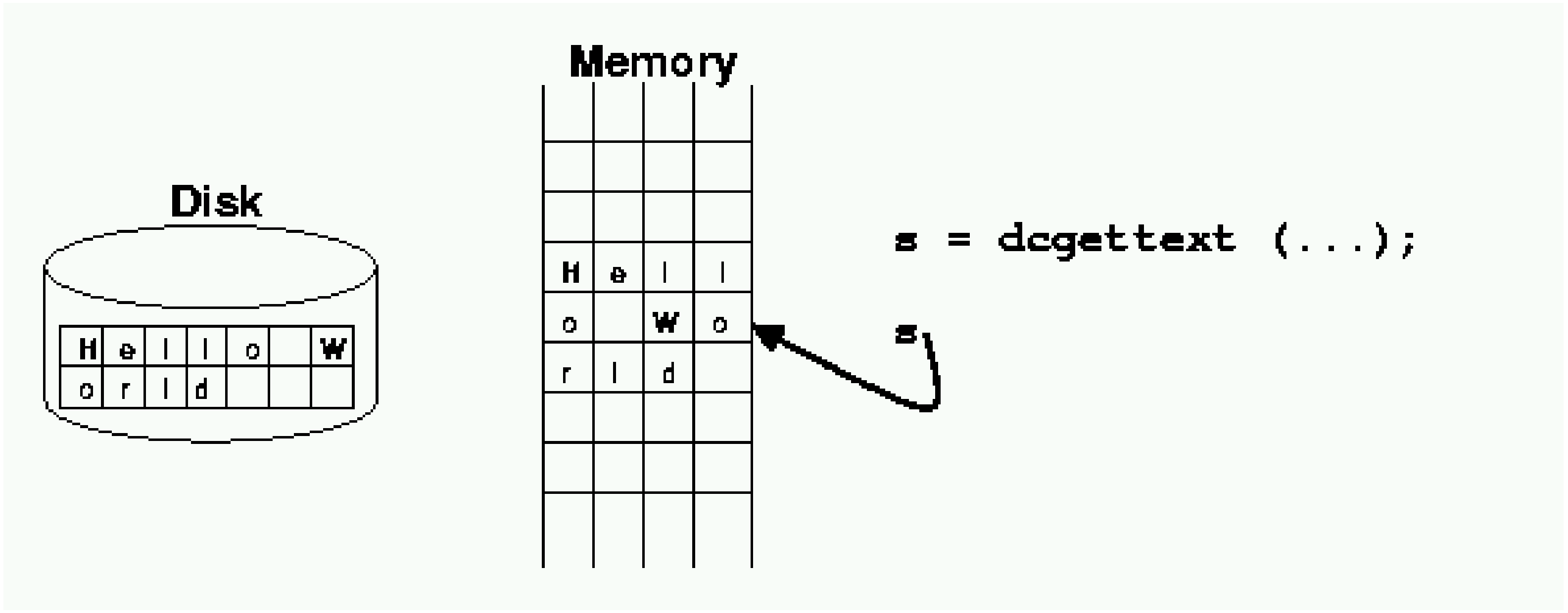
ISO 14651 Collation

- ☐ developed by experts based on POSIX.2
- ☐ much more flexible than the POSIX.2 model
- ☐ allows defining collation based on others (not only copying

On-The-Fly Conversion of Messages, I

Current situation:

- □ for every locale there must be a message catalog with the language and the character set of the locale

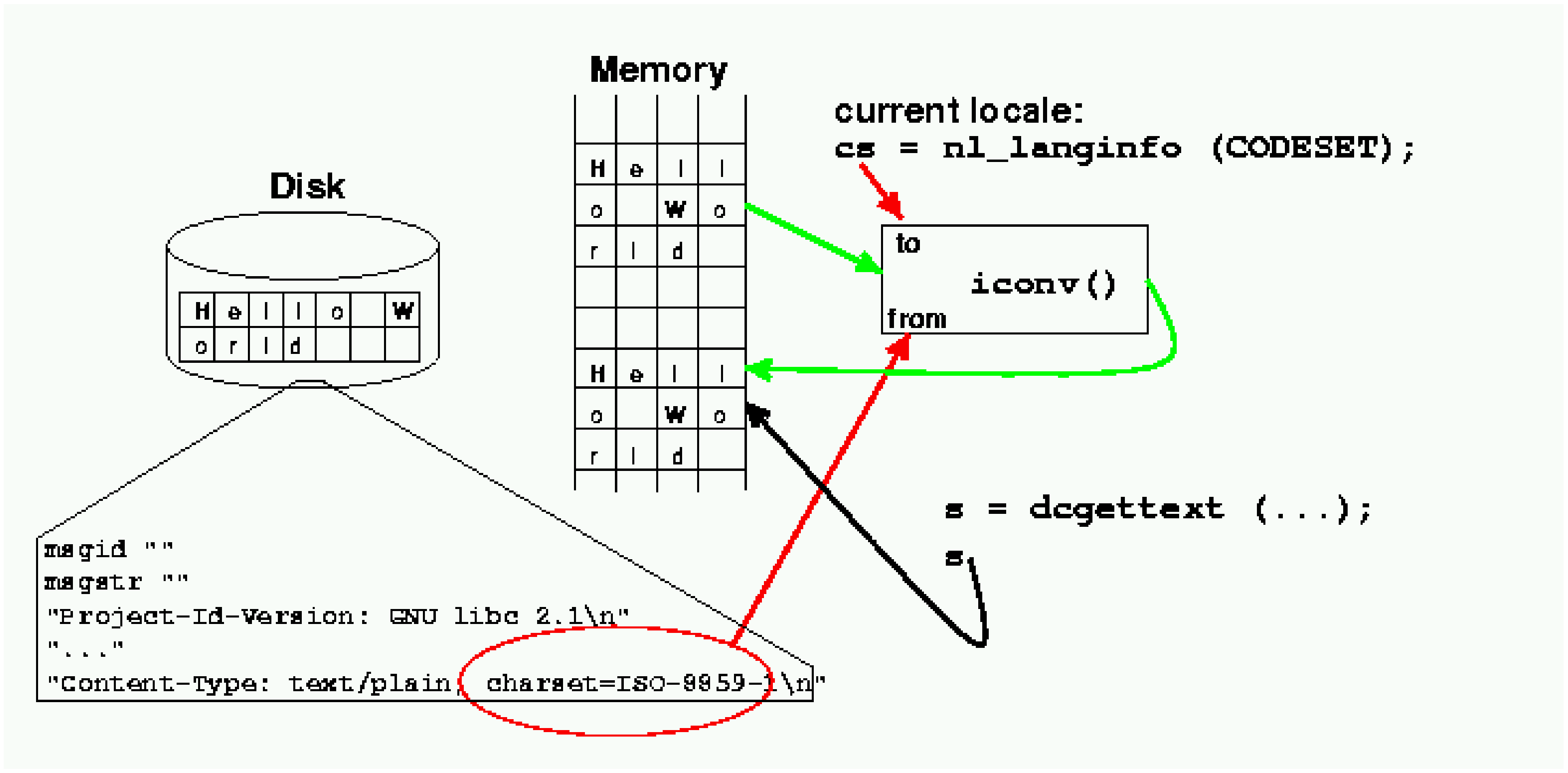


- □ too many files with the same information
- □ the catalog for the needed charset is not always available

On-The-Fly Conversion of Messages, II

New situation:

- the character set used for the messages is automatically adapted to the current locale



```
cs = nl_langinfo (CODESET);
```

```
msgid ""
msgstr ""
"Project-Id-Version: GNU libc 2.1\n"
"..."
"Content-Type: text/plain; charset=ISO-8859-1\n"
```

- only one catalog per language needed

Handling of Plural Forms, I

Plural forms of nouns are constructed differently in different languages

Old "Unix way":

```
    printf ("%d file%s deleted", n,
            n == 1 ? "" : "s")
```

Wrong for every language but English!!!

A bit better:

```
    printf (n == 1 ? "%d file deleted"
            : "%d files deleted", n)
```

Still works only for Germanic and some Asian languages.

Handling of Plural Forms, II

Slightly different in some languages:

- singluar form for 0 and 1 in some Romanic languages
- different form for 1, 2 and otherwise

Very complicated in some languages:

- singular for 1, first plural form for 2<=n%10<=4, second plural form otherwise
- singular for 1, first plural form for n%10==2, second plural form for 3<=n%10M=4, third plural form otherwise
- and several others

Consequence: hardcoding rules is no good idea!

Handling of Plural Forms, III

Implementation in glibc 2.2:

The translation files specify the number of plural forms and the formula to decide which one to use.

```
 nplurals=3; plural=n==1 ? 0 : n%10>=2&&n%10<=4 ? 1 : 2
```

New function

```
 dcngettext(const char *domain, const char *msgid1,
            const char *msgid2, unsigned long int n,
            int category)
```

allows specifying two message IDs; second only used for plural form if no translation is available

To-be-finished

Transliteration and Transcription Support

- The **`iconv()`** function currently stops when it finds an non-mappable character
- not always ideal
  - mail and news messages could live with information loss and transcribed text might be useful (e.g., romanji)
  - latin based languages often have replacement characters ('ae' instead of 'ä in German)
  - completely loosing the non-mappable characters also might be a solutio
- must be selectable by the user (not hardcoded!)

Transliteration

Implementation based on selectable modules

- The **`LC_CTYPE`** category contains information about
  - transliteration of single characters (not context sensitive)
  - ignorable characters
- This will be used in the default transliteration module
- user-installable modules also possible
- the modules will be used whenever a conversion stops because of an unmappable character
- modules will work on the UCS4 representation

Questions?

Comments?