I18n & L10n for PES

Internationalization
and
Localization
for PES

Kyle Nitzsche, 27 Feb 121013

Two kinds of presentations

This kind

The *other* kind

So let's just get on with it :)

All C and Python code here:

https://code.launchpad.net/~knitzsche/+junk/i18n

"Hello, world" terminal

```
$ export LANGUAGE=fr
$ gettext -e -s --domain=hello "hello,
world\n"
Bonjour, le monde.
$ export LANGUAGE=es
$ gettext -e -s --domain=hello "hello,
world\n"
hola, mundo
$ export LANGUAGE=zh_CN
$ gettext -e -s --domain=hello "hello,
world\n"
世界你好
```

hello-world.c

```
#include <stdio.h>
#include <libintl.h>
#include <locale.h>

int main(void) {

    setlocale(LC_ALL,""); // get env locale
    printf("%s\n", dgettext("hello", "hello, world\n"));

    return 0;
}
```

```
$ gcc -o it.o hello-world.c && LANGUAGE=de ./it.o
hallo, Welt

$ gcc -o it.o hello-world.c && LANGUAGE=pt_BR ./it.o
bom dia, mundo
```

hello-world.py

```
#!/usr/bin/python3
import locale
import gettext

# get env locale
locale.setlocale(locale.LC_ALL, '')
# translate
print(gettext.dgettext('hello', 'hello, world\n'))

$ LANGUAGE=zh_CN ./hello-world.py
世界你好

$ LANGUAGE=fr ./hello-world.py
Bonjour, le monde.
```

GNU gettext

- GNU gettext is the core of Ubuntu localization
- Provides the runtime libraries, APIs, tools and file formats
- Defines the files used in source packages, Launchpad Translations, and at run time

gettext has needs

- language
- locale
- domain
- msgid (the string being translated)
- (font)

What gettext needs to translate at runtime

- The current **language:** derived from the environment
- The **locale** for the language must exist locally (or the default 'C' locale is used, which is language unspecific)
- A gettext **domain**: needed to find the translation catalog
- **msgid**: the string being translated
- (An installed font that provides glyphs for the UTF-8 code points in the returned translation or else you see garbage glyphs)

Path to runtime translation

- Gettext uses the *language* and *domain* to find the path to the runtime translation file
- Language = fr
- Domain = hello
- *Standard* path to file:

/usr/share/**locale**/**fr**/LC_MESSAGES/**hello**.mo

- If not found there, gettext looks on the *language pack* path:

/usr/share/**locale-langpack**/**fr**/LC_MESSAGES/**hello**.mo

Path fallback is *essential*

- That gettext looks in two paths (standard and language pack) is essential to translations in Ubuntu and PES
- I'll cover this later in the presentation

Language

Gettext determines the current language through environment variables:

- **LANGUAGE** sets the preferred languages
  - LANGUAGE=zh_CN:en_GB:en
  - This fallback is on a per message basis (that is, per gettext call)
  - Most apps use LANGUAGE
- **LANG** sets the locale name: *ll_CC*.UTF-8
  - ll is the two letter language code
  - CC is the the two letter country code
  - For example: pt_BR.UTF-8 for Portuguese in Brazil
  - Mozilla apps still use LANG

Language specificity

- Most languages have one version used "everywhere"
  - For example, there's one translation of German, specified by it code: 'de'
- Some languages have different versions commonly associated with different countries, like Portuguese and Chinese
  - pt_BR (Portuguese Brazil)
  - pt (Portuguese)
  - zh_CH (Chinese* China)
  - zh_TW (Chinese Taiwan)

* Chinese language is Mandarin

Language defaults

System language defaults are set in:

- /etc/default/locale

```
$ cat /etc/default/locale
LANG="en_US.UTF-8"
LANGUAGE="en_US:en"
```

- Created at install time
- Modifiable later by user with "Language Support" app

Locale

- Locales are defined in **locales** package
- Cannot be used until instantiated on the system (default "C" locale used instead)
- Create a locale with:

```
sudo locale-gen ll_CC.UTF-8
```

  - *ll*: language code
  - *CC*: country code
  - UTF-8: the character encoding always used in Ubuntu

Locale, what is it

- A locale defines:
  - language*, country, charmap/encoding
  - number format (thousands sep char and decimal point char)
  - monetary symbols (three letter (EUR), symbol (€))
  - sort order ("collation")
  - paper size
  - time/date format
- * There's some schizophrenia on locale's LC_MESSAGES variable and LANGUAGE/LANG environment variables

Locales, showing

- Locales are created on language pack install
- Show the current locale with:

```
locale
```

- Show all currently defined locales with

```
locale -a
```

Locale, example

```
$ locale
LANG=en_US.UTF-8
LANGUAGE=en_US:en
LC_CTYPE="en_US.UTF-8"
LC_NUMERIC=en_US.UTF-8
LC_TIME=en_US.UTF-8
LC_COLLATE="en_US.UTF-8"
LC_MONETARY=en_US.UTF-8
LC_MESSAGES="en_US.UTF-8"
LC_PAPER=en_US.UTF-8
LC_NAME=en_US.UTF-8
LC_ADDRESS=en_US.UTF-8
LC_TELEPHONE=en_US.UTF-8
LC_MEASUREMENT=en_US.UTF-8
LC_IDENTIFICATION=en_US.UTF-8
LC_ALL=
```

Locales, supported

- Locales *theoretically* supported in Linux:
  - /usr/share/i18n/SUPPORTED
- Locales *actually* supported in ubuntu/ubiquity, in ubiquity source: d-i/source/localechooser/languagelist
- Modify that file to set language's displayed to end user at installation (ubiquity)
- It is usually safe to dpkg-divert that file: it is deleted with ubiquity after installation

Locale, obtain in code

- Python:

```
import locale
locale.setlocale(locale.LC_ALL, '')
```

- C:

```
#<include locale.h>
..
setlocale(LC_ALL,'');
```

Gettext domains, packaging

- Domain is usually the **package name**, for example *gedit*, and is set in the source package (and used in code modules)
- In autotools packages, domain is often set in configure.ac:

  ```
  AC_SUBST([GETTEXT_PACKAGE], [gedit])
  ```
- In python distutils packages it is often set in setup.cfg:

  ```
  domain=language-selector
  ```

Gettext domain, can use others

- Code can call other domains explicitly
- For example, many packages use stock gtk widgets (in addition to their own GUI)
- gtk stock widgets translations are provided by *gtk20* and *gtk30* domains
- These domains are installed by language packs

Domain, setting in code

- One usually sets the default domain in a code module at the beginning
- If set, later calls to gettext can omit the domain using the simple gettext(..) call
- If not set, you must specify the domain on every gettext call
- One can obtain translations from specific domains by specifying the domain with **d**gettext(domain, msgid) style calls
- There are several variants of 'd' gettext

domains.py

```
#!/usr/bin/python3

import locale
import gettext

# get env locale
locale.setlocale(locale.LC_ALL, '')
# set domain
gettext.textdomain('hello')
# translate
print(gettext.gettext('hello, world\n'))

$ LANGUAGE=ja ./domains.py
こんにちは、世界
```

domains.c

```
#include <stdio.h>
#include <libintl.h>
#include <locale.h>

int main(void) {
    setlocale(LC_ALL,""); // get env locale
    textdomain("hello");
    printf("%s\n", gettext("hello, world\n"));
    return 0;
}
$ gcc -o it.o hello-world.c && LANGUAGE=de ./it.o
hallo, Welt
```

Language, changing in code

- As noted, the app display language is usually that of the environment: LANGUAGE var
- You can change the display language by changing the env variable in code, then calling setlocale()

language.c

```
#include <stdio.h>
#include <stdlib.h>
#include <libintl.h>
#include <locale.h>
int main(void) {
    char * msgid = "Type a file name";
    setlocale(LC_ALL,""); // get env locale
    textdomain("gtk20"); // gettext domain
    char * trans = gettext(msgid);//get trans
    printf("msgid: %s\n", msgid);
    printf("Translation in env language:\n\t%s\n", trans);
    setenv("LANGUAGE", "es", 1);// change to Spanish
    setlocale(LC_ALL,"");
    trans = gettext(msgid);
    printf("Translation in Spanish:\n\t%s\n", trans);
    return 0;
}
```

language.c, continued

```
$ gcc -o it.o language.c && LANGUAGE=pt_BR ./it.o
msgid: Type a file name
Translation in env language:
   Digite um nome de arquivo
Translation in Spanish:
   Teclee un nombre de archivo
```

language.py

```
#!/usr/bin/python3
import locale
import gettext
import os

locale.setlocale(locale.LC_ALL, '') # get env locale
gettext.textdomain('gtk20') # gettext domain
msgid = 'Type a file name'
print('msgid: ', msgid)
print("Initial language: ", os.environ['LANGUAGE'])
print('\tTranslation:', gettext.gettext(msgid)) #
translate
os.environ['LANGUAGE'] = 'es'
print("New language: ", os.environ['LANGUAGE'])
print('\tTranslation', gettext.gettext(msgid)) # translate
```

language.py, continued

```
$ LANGUAGE=de ./language.py
msgid:  Type a file name
Initial language:  de
   Translation: Geben Sie einen Dateinamen an
New language:  es
   Translation Teclee un nombre de archivo
```

Number display is localized

- The decimal point char varies by language
- The thousands separator char varies by language
- Set the locale's LC_NUMERIC var to change these
- Then, in C, these chars are available from a struct returned by localeconvert()

numbers.c

```
int main(void) {
    char * locale = setlocale(LC_ALL,"");
    struct lconv * lloc; //struct of locale values, including numeric
    char * locs[8] = {"da_DK.UTF-8", "de_DE.UTF-8", "en_US.UTF-8",
                     "fr_FR.UTF-8", "ja_JP.UTF-8", "pt_BR.UTF-8",
                     "ru_RU.UTF-8", "zh_CN.UTF-8"};
    double num = 12345.67;
    int i;
    for ( i = 0; i < sizeof(locs)/sizeof(char *); i++ ) {
        setlocale(LC_NUMERIC, locs[i]);
        lloc = localeconv();
        int thous = (int){num/1000};
        int hund = (int){num - (thous * 1000)};
        int dec = (int){((num - (thous * 1000)) - (hund)) * 100};
        char * dsep = lloc->decimal_point;
        char * tsep = lloc->thousands_sep;
        printf("%s\t%5.2f\t", locs[i], num);//decimal char used, but not thous sep
        printf("%d%s%d%s%d\n", thous, tsep, hund, dsep, dec);
    }
    return 0;
}
```

numbers.c, continued

```
$ gcc -o it.o  numbers.c ./it.o
(local          printf     manual)
da_DK.UTF-8  12345,67   12.345,67
de_DE.UTF-8  12345,67   12.345,67
en_US.UTF-8  12345.67   12,345.67
fr_FR.UTF-8  12345,67   12 345,67
ja_JP.UTF-8  12345.67   12,345.67
pt_BR.UTF-8  12345,67   12.345,67
ru_RU.UTF-8  12345,67   12 345,67
zh_CN.UTF-8  12345.67   12,345.67
```

numbers.py

```
#!/usr/bin/python3
import locale
import gettext

locale.setlocale(locale.LC_ALL, '') # get env locale
locs = ["da_DK.UTF-8", "de_DE.UTF-8", "en_US.UTF-8",
           "fr_FR.UTF-8", "ja_JP.UTF-8", "pt_BR.UTF-8",
           "ru_RU.UTF-8", "zh_CN.UTF-8"]
for loc in locs:
    locale.setlocale(locale.LC_NUMERIC, loc)
    print(loc, locale.format('%.2f',12345.67))
```

numbers.py, continued

```
$ ./numbers.py
da_DK.UTF-8 12345,67
de_DE.UTF-8 12345,67
en_US.UTF-8 12345.67
fr_FR.UTF-8 12345,67
ja_JP.UTF-8 12345.67
pt_BR.UTF-8 12345,67
ru_RU.UTF-8 12345,67
zh_CN.UTF-8 12345.67
```

Currency is localized

- Every currency has a symbol: €, ￥, $, etc
- And an international abbreviation: EUR, CNY, USD
- Set the locale's LC_MONETARY var to change these
- Then, in C, these chars are available from the same struct returned by localeconvert()

currency.c

```
#include <stdio.h>
#include <string.h>
#include <locale.h>
int main(void) {
    setlocale(LC_ALL,"");
    char * locs[8] = {"da_DK.UTF-8", "de_DE.UTF-8", "en_US.UTF-8",
"fr_FR.UTF-8", "ja_JP.UTF-8", "pt_BR.UTF-8", "ru_RU.UTF-8", "zh_CN.UTF-
8"};
    int i; double num = 12345.67;
struct lconv * lloc;
    printf("\nLOCALE\t\tINT CURR ABBR\tSYMBOL\tNUMBER\t\tMONEY\n");
    for ( i = 0; i < sizeof(locs)/sizeof(char *); i++ ) {
        setlocale(LC_NUMERIC, locs[i]);
        setlocale(LC_MONETARY, locs[i]);
        lloc = localeconv();
        printf("%s\t%s\t\t%s\t%5.2f\t%s%5.2f\n", locs[i],
lloc->int_curr_symbol, lloc->currency_symbol, num, lloc->currency_symbol,
num);
    }
    return 0;
```

currency.c

```
$ gcc -o it.o currency.c && ./it.o

LOCALE       INT CURR ABBR    SYMBOL   NUMBER       MONEY
da_DK.UTF-8  DKK               kr       12345,67        kr12345,67
de_DE.UTF-8  EUR              €         12345,67    €12345,67
en_US.UTF-8  USD              $         12345.67    $12345.67
fr_FR.UTF-8  EUR              €         12345,67    €12345,67
ja_JP.UTF-8  JPY              ￥        12345.67        ￥12345.67
pt_BR.UTF-8  BRL              R$        12345,67    R$12345,67
ru_RU.UTF-8  RUB              руб       12345,67    руб12345,67
zh_CN.UTF-8  CNY              ￥        12345.67    ￥12345.67
```

msgid and *msgstr*

- *msgid* is the gettext name for the untranslated string
- *msgstr* is the gettext name for the translated string

Plurals

- Languages handle plurals in surprisingly different ways
- In English, we use the *plural* noun with *zero* of something (yes, weird)
  - "I have zero car**s**"
- Chinese has no plurals; Russian has three forms; etc.
- The gettext API and the toolchains support plurals for most languages
- Developers must use the API for quality results

Plurals, continued

- Use **n**gettext (and d**n**gettext and dc**n**gettext)
- '**n**' for *number*
- ('d' we have seen is for domain)
- In code, provide a singular and a plural string
- And the *number*, not known till run time

```
ngettext("Found and replaced %d occurrence",
"Found and replaced %d occurrences",
occurrences)
```

Translators and toolchain take care of the rest

plural.c

```
#include <stdio.h>
#include <libintl.h>
#include <locale.h>
void doit(num) {
    printf( ngettext( "Opening %d Item",
        "Opening %d Items", num), num);
// NOTE num has to be passed to printf too or there's a string literal
error
    printf("\n");
}
int main(void) {
    int num = 0;
    setlocale(LC_ALL,"");
    textdomain("gtk30");
    doit(0);
    doit(2);
    doit(21);
    return 0;
}
```

plural.c, stdout

Russian has three plural forms:

```
$ gcc -o it.o plural.c && LANGUAGE=ru ./it.o
Открывается 0 элементов
Открывается 2 элемента
Открывается 21 элемент
```

French has two plural forms:

```
$ gcc -o it.o plural.c && LANGUAGE=fr ./it.o
Ouverture de 0 élément
Ouverture de 2 éléments
Ouverture de 21 éléments
```

Chinese has zero forms (does not distinguish this way):

```
$ gcc -o it.o plural.c && LANGUAGE=zh_CN ./it.o
打开 0 项
打开 2 项
打开 21 项
```

plural.py

```
#!/usr/bin/python3

import locale
import gettext

# get env locale
locale.setlocale(locale.LC_ALL, '')
# set domain
gettext.textdomain('gtk30')

def doit(num):
    print(gettext.ngettext( "Opening %d Item", "Opening %d Items", num) %
num)
    // Note, you need to pass num to the print statement too
doit(0)
doit(1)
doit(21)
```

plural.py stdout

Russian:

```
$ LANGUAGE=ru ./plural.py
Открывается 0 элементов
Открывается 1 элемент
Открывается 21 элемент
```

French:

```
$ LANGUAGE=fr ./plural.py
Ouverture de 0 élément
Ouverture de 1 élément
Ouverture de 21 éléments
```

Chinese:

```
$ LANGUAGE=zh_CN ./plural.py
打开 0 项
打开 1 项
打开 21 项
```

gettext function names

Usually you will see gettext functions in C and Python as short hand names:

- `gettext() = _()`
- `ngettext() = n_()`
- `dgettext() = d_()`

gettext function names python

```
>>> from gettext import gettext as _
>>> gettext.textdomain('gtk20')
'gtk20'
>>> _('Type a file name')
'Teclee un nombre de archivo'
>>> import locale
>>> import os
>>> os.environ['LANGUAGE']='ja'
>>> from gettext import gettext as _
>>> gettext.textdomain('gtk20')
'gtk20'
>>> _('Type a file name')
'ファイル名を入力してください'
```

gettext function names C

```
#include <stdio.h>
#include <libintl.h>
#include <locale.h>
#define _(STRING) gettext(STRING)
int main(void) {
    setlocale(LC_ALL,""); // get env locale
    textdomain("gtk20");
    printf("%s\n", _("Type a file name"));
    return 0;
}
$ gcc -o it.o  gettext_names.c && LANGUAGE=ru ./it.o
Введите имя файла
$ gcc -o it.o  gettext_names.c && LANGUAGE=de ./it.o
Geben Sie einen Dateinamen an
```

Collecting the translatable strings: *pot* file

- The source tree/source package is scanned
- Strings wrapped in standard gettext patterns and others are found
- Placed in a single file: the *po template* file ("pot")
- The *pot* file contains the *current set of translatable messages*
- debhelper `dh_translations` (debian/rules) handles most normal file types where i18n strings are wrapped in standard gettext

Source packages: po/ dir

- Source packages have a /po directory for translations
- Contains:
  - a (generated) *domain*.pot file
  - *language*.po files for each supported language
  - Typically a POTFILES.in that lists files with translatable messages
- There can be multiple po directories in different places
- debconf templates are a special case

Sample po/ directory

```
$ ls po
af.po   ca.po            en_CA.po  fur.po  id.po  ln.po   nb.po          pt_BR.po  sv.po   uz.po
am.po   ca@valencia.po   en_GB.po  fy.po   is.po  lo.po   nds.po         pt.po     szl.po  vec.po
an.po   ckb.po           eo.po     ga.po   it.po  lt.po   ne.po          ro.po     ta.po   vi.po
ar.po   crh.po           es.po     gd.po   ja.po  lv.po   nl.po          ru.po     te.po   zh_CN.po
ary.po  csb.po           et.po     gl.po   jv.po  mg.po   nn.po          sc.po     th.po   zh_HK.po
ast.po  cs.po            eu.po     gu.po   ka.po  mhr.po  oc.po          sd.po     tk.po   zh_TW.po
az.po   cv.po            fa_AF.po  gv.po   kk.po  mi.po   os.po          se.po     tl.po
be.po   cy.po            fa.po     he.po   km.po  mk.po   pam.po         shn.po    tr.po
bg.po   da.po            fil.po    hi.po   kn.po  ml.po   pa.po          si.po     trv.po
bn.po   de.po            fi.po     hr.po   ko.po  mn.po   pl.po          sk.po     tt.po
bo.po   dv.po            fo.po     ht.po   ku.po  mr.po   POTFILES.in    sl.po     ug.po
br.po   el.po            fr.po     hu.po   ky.po  ms.po   POTFILES.skip  sq.po     uk.po
bs.po   en_AU.po         frp.po    hy.po   lb.po  my.po   ps.po          sr.po     ur.po
```

No *domain*.pot file yet

POTFILES.in lists source files to consider

Update pot: `intltool-update -p`

- When the .po/POTFILES.in is present, enter the /po directory and
  - `intltool-update -p`
  - You may want to specify the generated pot file name with -g
- There are other methods, for example: make
- Once the pot file is up to date, po files can be updated from it, although this **must only be done at the right time** to avoid accidentally dropping translations

pot file: PES responsibilities

- PES provides translation services at Project Management request
- Previously, whether this included i18n was not fully clear
- This is now clarified: it is the responsibility of the engineering/project team **to ensure when the pot file is updated, it includes all strings subject to translation**
- PES translation services will update the po files from the pot file, get them translated, and return them
- **Run time display of translated strings is also clearly now an engineering/project responsibility**

Source pkg: po files

- One per language of translation: for example pt.po and pt_BR
- Header section contains key data:
    - character encoding (must always be UTF-8
    - Translator credit
    - Language
    - Plural forms for language
- The body contains msgid/msgstr stanzas
- A comment before each stanza shows the translation's source code file and line number

Fuzzy translations

- Common source of confusion
- When a source string changes a little, the existing translation may only need a small modification
- It is marked *fuzzy* in the po file
- Unless modified and fixed, it is **not valid and not used**
- This mechanism prevents unnecessary work by translators
- They can reuse most of the existing translation

Fuzzy example

Example

```
#: ../libnautilus-private/nautilus-file-operations.c:5149
#, fuzzy
msgid "Setting permissions"
msgstr "Stel regte"
```

Caution: Do not update po files until final string freeze, or you may lose translations due to fuzzy marking

mo files

- mo files are the run time translation catalogs
- *domain*.mo
- sorted by directory
- Examples:

/usr/share/locale-langpack/**fr**/LC_MESSAGES/gedit.mo
/usr/share/locale-langpack/**es**/LC_MESSAGES/gedit.mo

See what is in it with `msgunfmt -o OUTFILE FILE`

gettext *search path*

As noted, gettext (in Ubuntu) looks in two places for translations:

- First, in the *standard* install location:
  - /usr/share/**locale**/..
- Then, in the *language pack* install location:
  - /usr/share/**locale-langpack**/..

This is combined with **translations stripping** in Ubuntu and has importance to PES projects

Why two paths?

gettext's search path fallback allows:

- Modified Ubuntu packages and upstream packages to deliver their own translations
- While stock Ubuntu packages fall back to language packages

This **depends on** translation stripping in Ubuntu, as described next

Ubuntu translations are stripped from packages

Debian packages normally install all their translations

- User gets **all languages** even though they may only want one -- disk **waste**

In Ubuntu and most PES PPAs (main, and restricted), translations are **stripped** from the binary packages at build time (due to pkgbinarymangler installed and configured in Ubuntu builders and PES PPAs)

- Translations are installed by language packs

In PES PPAs, translations should usually be stripped

- Most packages in PES **do not modify messages** exposed to the user
- They *should* have their translations stripped like Ubuntu packages **if** the package is a fork from a package in main (or restricted)
- Or else, the package will install upstream translations from the source package (not Ubuntu translations from the community & language package) and **they will be incomplete and not updatable from community/Ubuntu**
- PES PPAs were set up to do this stripping
- It can be disabled for a package in its debian/rules

Disable translation stripping

If you modify user exposed strings in a package that is forked from Ubuntu (main/restricted) and you want it translated:

- You need to get the latest correct translations from Launchpad and put them in your source package
- Disable translation stripping by adding this early to the debian/rules file:

```
export NO_PKG_MANGLE=anything
```

Set it to any value to disable stripping for the package

PES Translation Service (PTS)

- PES obtains translations from vendors
- Competitive bidding
- Defined process
- Defined responsibilities

PTS does translation *only*

Going forward, the project engineering team is responsible for i18n/l10n of source code, including:

- Ensuring strings are carried into the pot file (at String Freeze)
- Run time translations work

The PTS service is responsible for obtaining translations for specified packages' pot files for the target languages in a competitive bidding process