Using `glibc 2.1`

Ulrich Drepper*
Cygnus Solutions
Sunnyvale, CA

March 1st, 1999

*drepper@cygnus.com

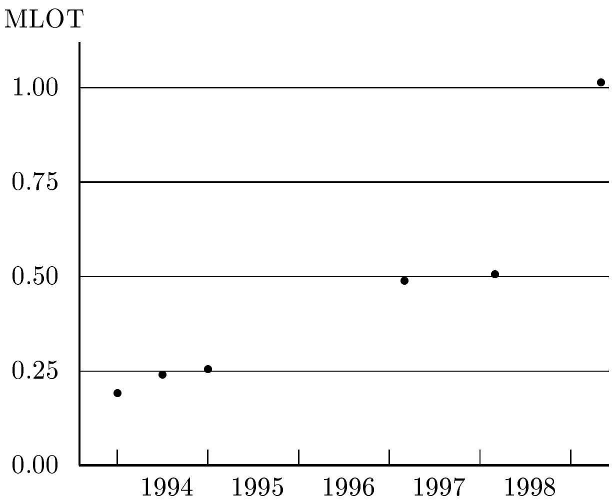
Slide 1

History

- development started in 1991, mainly by Roland McGrath
- first somewhat usable releases in 1992
- at about this time HJ Lu splits up the Linux libc
- parallel development
  - fast, but less controlled Linux libc development. It settled first in libc4 (`a.out`), later libc5 (ELF).
  - slower, but clean and safe development of GNU libc
- January 1997: release of glibc 2.0. The goal was to catch up with the Linux libc5.
- February 1999: release of glibc 2.1. Compliant with all important standard, well tested and ported to all Linux platforms.

The first years of the development were marked by slow progress. Roland McGrath was employed by the FSF to write the GNU system but the libc was only one part. At that time the other needed tools were not yet in a shape adequate for general use. Even GNU `make` had yet to be finished. Many of the (sometimes exotic) GNU `make` extensions are due to the fact that they are used in the GNU `libc` makefiles.

When the Linux kernel became useful in 1992 a C library was needed. Somebody had to step forward to do the work of putting existing code together and write the Linux-specific portions. This someone was HJ Lu who took the existing pieces of the GNU `libc` and added missing functions from all possible sources to make it usable.



One interesting thing about the development might be to see the development of the size of the distribution. The graph at the left shows the number of million lines of text (MLOT, I don't count lines of code since the library does not only consists of code). It shows that the size of `glibc 2.0` was approximately twice the size of the previous release (`glibc 1.09.1`) and that `glibc 2.1` again doubled in size from the 2.0.6 release. This trend will not continue in this way, though.

Slide 2

What's new?

`glibc 2.1` introduces news in seven categories:

- add to compliance with Unix 98 (Single Unix Specification)
- more POSIX.1 and POSIX.2 support
- library extensions from ISO C9x.
- IPv6 interfaces are added
- NIS+ and secure RPC support
- new kernel interfaces
- non-standard extensions

Beside this of course lots of bug fixes went in.

In the remainder of this document the new interfaces and if necessary extensions to existing interfaces will be described. The total number of interfaces went up from 1186 in `glibc 2.0` to 1521 in `glibc 2.1`. Split up into the seven groups it reads as follows:

Unix98: 69 new interfaces. Some are dummy implementations of the STREAMS interface. Another big group implement the LFS (Large File Support) interface.

POSIX: The POSIX.2 interface is now complete and the Linux kernel implements the POSIX.1b realtime signal interface. In total these are six new interfaces.

ISO C9x: The upcoming ISO C9x standard introduced in the library part a lot of new functions. Many were already implemented in `glibc` but still 140 new functions were added.

IPv6: Coming from the latest draft of the IPv6 interface specification we have nine new functions.

Secure RPC: 40 new functions to implement the Secure RPC interface, as available in Solaris.

Kernel: New kernel functionality required 5 new functions. Note that these are only those functions not found in any standard and therefore requiring a non-standard interface.

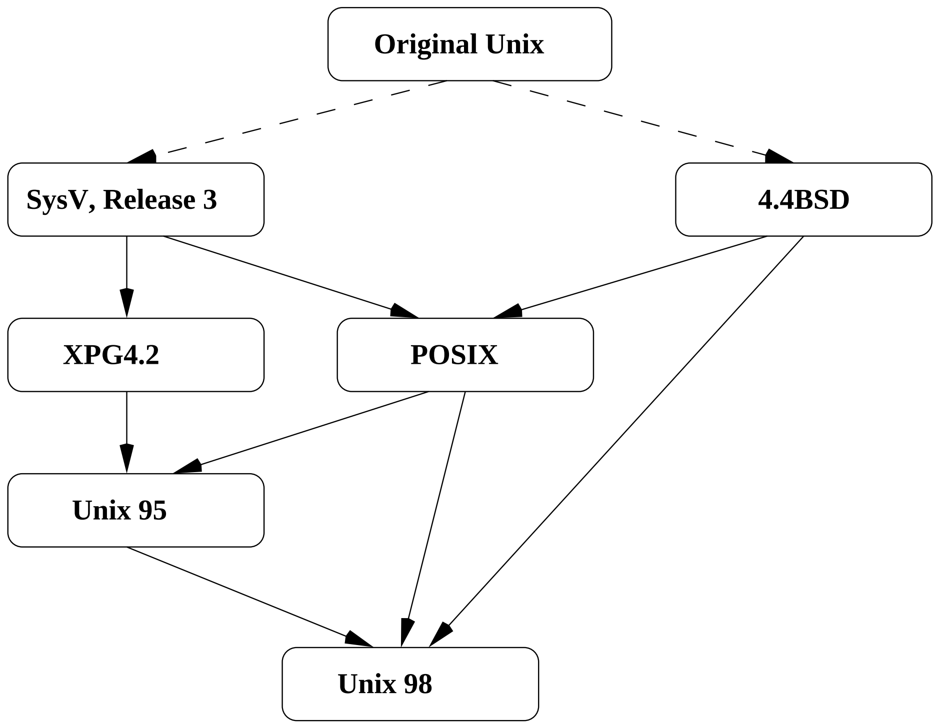
GNU extension: We have 75 new interfaces which are not part of any standard but nevertheless were found useful.

Some words on standards and the selection of them

The `glibc` follows several different standards which include:

- 4.4BSD
- System V, release 3
- X/Open Portability Guide 4
- POSIX.1 and POSIX.2
- Unix 95
- Unix 98 (Single Unix Specification, version 2)

ISO C is left out of this since it does not cover OS interfaces. The incomplete heritage is somewhat like this:



Code is often written for a specific environment in mind and since different environments are often incompatible it should be possible to switch back and forth between these environments. This is what feature select macros are for. `glibc` supports a variety of them, some of them being defined in standards, others are GNU inventions. To use a feature select macro one has to define this macro to a numeric value before including the first system header file. Preferrably this happens on the compiler commandline. Normally any value work, but there are a few macros which allow switching between multiple standards and therefore depend on certain value. The feature select macros known in `glibc 2.1` are:

_BSD_SOURCE for BSD specific code, 4.4BSD and before.

_SVID_SOURCE for System V code, release 3 and before.

_POSIX_SOURCE for using POSIX.1 functionality.

_POSIX_C_SOURCE this macro has different effects depending on the value. If the value is

- 1 it has the same effect as _POSIX_SOURCE.
- >= 2 POSIX.2 functionality is added.
- >= 199309 the extensions from POSIX.1b are added, and
- >= 199506 the POSIX.1c extensions are added.

_XOPEN_SOURCE_EXTENDED for XPG4.2 functionality.

_XOPEN_SOURCE Again a macro which has different effects depending on the value. If the value is

- not 500 the XPG definitions from revision 1 to 4 are used (not 4.2). This includes POSIX.1 (not .1b and .1c) and POSIX.2 as well as much of the System V functionality.
- 500 Unix 98 functionality is selected. This includes (as the diagram shows) all of XPG4.2, Unix98, POSIX, and quite a lot from BSD.

_LARGEFILE64_SOURCE this is an official extension to Unix 98 which adds functionality to support support an alternative interface to most filesystem operations. We will see this later.

_FILE_OFFSET_BITS There are two value this macro can have:

- if it is 32 nothing changes.
- if it is 64 on 32bit systems the filesystem operations like `open` are transparently replaced by functions which can handle files of length greater than 2^{31} bytes.

_GNU_SOURCE this is the most completely set of functionality. it includes Unix 98 and several GNU extensions.

The question is which subset of the C library functionality should be used. The answer is: it depends.

The decision must be made based on the target list. If all platforms are modern Unices the base standard should definitely be Unix 98 as it provides the longest list of functionality. If this is not a possibility restricting to POSIX functions is probably the second best choice. The missing functionality then should be provided by the application itself.

But this is only one possibility to attack this problem. Another possibility is how GNU programs are written. They always use the maximum set of functionality from the libraries. This means for `glibc` to use _GNU_SOURCE. For all non-standard functions the programs provide replacements and using tests for all the functionality at compile time it is possible it can be determined which replacements have to be used. This has the advantage that the maximum number of functions from the libraries can be used which leads to reduced code size (due to the use of shared objects) and higher performance (since the native libraries are in general faster than the generic implementations).

So my preoccupied advice is to follow the GNU approach and use the configuration tools (`autoconf`, `automake`, and `libtool`) to use as much functionality from the target system as possible. This means the feature select macros are not really important. For existing code they are important, though, since not only it could come to namespace problems when the wrong feature set is selected, some functions have a different behaviour depending on the standard to be followed. So in this case careful study of the existing source code must precede the compilation attempts.

Slide 3

Unix 98 extension

Most complete standard, combining POSIX, SysV, and BSD.
`glibc 2.0` missed a number of function families:

- `utmpx` function, to keep track of logons.
- STREAMS.
- `iconv`, character set transformation functions.
- simplified signal handling.
- System V pseudo terminal handling.
- LFS (Large File Support), this is an Unix 98 extension.
- miscellaneous functions.
- legacy interfaces.

The Unix 98 standard evolved from the previous standards by the OpenGroup. They were in sequence XPG (X/Open Protability Guide), version 1, 2, 3, 4, and 4.2. Then the OpenGroup got the rights of the Unix brand and XPG 4.2 evolved into Unix 95 which finally ended in Unix 98. It is often called "Spec 1170" since it defined at some time 1170 interfaces (it is not true anymore).

The OpenGroup is making this standard available in two forms:

- The book *Do Solo 2* by the OpenGroup contains the interface definition in brief form in the book. The CD coming with the book contains the complete standard in HTML form. While the printed information is not really useful (it is far too brief) the HTML documentation is excellent. It also covers the latest POSIX standards and therefore makes buying the expensive POSIX standard unnecessary unless one needs very detailed information from the later.

  Go Solo 2, The authorized Guide to Version 2 of the Single Unix Specification; The OpenGroup, Prentice Hall, Englewood Cliffs, NJ 07632. ISBN 0-13-575689-8, Document Number X909P.

- Via the Internet, available to registers users. The registration is free. Visit

  `http://www.db.opengroup.org/sud/`

  to get access to the documentation.

While initially being an industry standard and therefore excluding the development advances of the BSD Unix variants Unix 95 incorporated a lot of the POSIX standards available at that time. Unix 98 did one further step and not only added the missing POSIX functionality but also added some of the BSD functionality which had no equivalent in the Unix 95 standard. This is why one can say Unix 98 embraces all the standards which on the one hand is good since now almost all of the needs of the programmer are standardized. On the other hand the size of the library increased dramatically. But since this development now comes to a rest this should not be a real problem.

Slide 4

`utmpx` functions

- `utmp` functions were first available in BSD system, used to log user and system activities such as login or system starts.
- added to Unix 95, but with a different, extended interface (`utmp` became `utmpx`)
- `glibc 2.0` contained the `utmp` interface, but in a form not suitable for IPv6.
- `glibc 2.1` contains a new implementation of the `utmp` interface which is at the same time available under the new names from the `utmpx` interface.

Problem: How to solve the problems with binaries using older libraries (`libc 5`, `glibc 2.0`) using the same files?

What is `utmpx` good for?

The `utmpx` interface (or the `utmp` interface, for that matter) is not very prominent since the normal programmer does not often use it. It is almost exclusively used by system utilities. But as soon as something breaks the consequences are widely visible.

To understand this a brief introduction is adequate. System utilities often have to know whether a user is logged in and if yes, how s/he can be reached. This is not much of a problem in a single-user environment but it is in Unix environments.

Therefore programs like `login` "announce" new users by writing information about the login into a file. Other information such as reboot times, system reboots, runlevel or system time changes are also added to put the logins into relation to the events on the system. E.g., this helps to identify a crashing system. Since all user processes are lost at the time a system crashes and there is no time before the crash to remove the login entry from the bookkeeping file we have to have another way. Marking system starts helps since this information also includes the information that all previously logged in users are now not logged in anymore.

Two examples should show that the collected information is important

- All logins are logged including the information who logged in from which place. This might help a user to recognize intruders. If suddenly `telnet` reports that the last login happened from a strange host, the system might have been cracked.
- Each user is logged in with the terminal s/he is available at. The once very popular `talk` programs used this information to select a terminal where a certain user can be found.

The file format was chosen to consist of records of fixed width. This has two consequences: it is hard to extend the dumped data structure, and related to this, if the data structure changed programs using the same file but different record length will mess up the file. But this is the situation on Linux systems where all, `libc 5`, `glibc 2.0` and `glibc 2.1` all have different record lengths.

Slide 5

Solution: Automatic Conversion

`glibc 2.1` contains the `utmpd` (`utmp` Daemon) which helps to solve this situation.

- It automatically detects changes to the `utmp`, `utmpx`, `wtmp` and `wtmpx` files.
- If these changes are made in a different but understood format it automatically converts to the new formats.
- All changes are propagated to the old (e.g., `utmp`) and new (`utmpx`) file.
- Programs using `glibc 2.1` talk to the daemon directly.

The `utmp` daemon was mainly implemented by Mark Kettenis `kettenis@phys.uva.nl`. It was done because after the release of `glibc 2.0` one of the common complains was about trashed `utmp` and `wtmp` files because of programs using `libc 4` or `libc 5` at the same time.

The implementation of the `utmp` handling functions in `glibc 2.1` can work directly with the files or indirectly via the daemon. Each program which needs any of these functions finds first out whether the daemon is running or not. If it is running the daemon is used, otherwise (obviously) not.

This means that people who do not have old programs lying around safely can avoid running yet another daemon and everything works fine. If you have programs with corruptions of the `utmp` files simply start the daemon at startup time and you should be fine from now on.

The program is normally in `/usr/sbin/utmpd`. It requires no commandline parameters so putting

```
/usr/sbin/utmpd
```

in one of the startup files does the whole trick.

Slide 6

STREAMS

- The Linux kernel does not support STREAMS.
- There are third-party patches to add STREAMS support (Caldera).
- But there is also a promising user-level STREAMS implementation which I hope some day to support in `glibc`.
- The STREAMS interfaces required in Unix 98 are available in `glibc 2.1` but they are all stub implementation which always fail.
- But they can be used in programs which fall back on BSD sockets if STREAMS are not available.

Why STREAMS?

The short answer: I don't know. The longer one: the STREAMS interface is part of the Unix 98 specification. This is no big surprise as it was developed for System V release 3 and ever since comes with systems derived from this. Developers got told this is the future of network programming and some were foolish enough to believe it.

A short "introduction" into STREAMS: AT&T introduced in the 8th edition of Unix the stream based I/O system. Now instead of talking to a device driver (real or virtual) through character I/O system, e.g. sockets, one communicates with an module which can be implemented in many ways. The user has the possibility to use other modules to talk to the same device or even stack arbitrary many modules on top of each other. Things like customizing device drivers which has to happen through magic `ioctl` calls or `proc`-filesystem operations now can be implemented in a much more elegant way by special STREAMS modules.

This all sounds fine but it is not without problems: the STREAMS stack implementation is far too slow. It already became obvious when faster machines used 10Mbit Ethernet for transformation. The multiple copy operations of the data which is passed from one module to another reduced the performance dramatically. This is even more true with today's high-end network technologies (Gigabit Ethernet, ATM).

For these reasons commercial Unices using STREAMS also again provided normal BSD sockets as the fast alternative and programmers get back to use them instead of STREAMS. But still there is a lot of legacy code out there which with the rise of Linux has to be ported.

The plan for `glibc` is to provide a user-level STREAMS implementation. At least up to the level where it is possible. There are people working on this but I am not aware of the state of affairs.

To support the STREAMS interface `glibc` now provides the header `stropts.h` which declares the following interfaces:

`isastream` Test a file descriptor for being a STREAM.

`fattach` Attach a STREAMS-based file descriptor to a file in the file system name space.

`fdetach` Detach a name from a STREAMS-based file descriptor.

`getmsg`

`getpmsg` Receive next message from a STREAMS file.

`putmsg`

`putpmsg` Send a message on a STREAM.

`ioctl` Control a STREAMS device.

Only the `isastream` function is implemented so that it can be used. It simply returns always zero which indicates that the file descriptor is not associated with a STREAM. Of course this is always correct. The `ioctl` function can be used on descriptors other than those associated with STREAMS. All other functions simply will fail all the time.

References for STREAMS:

Unix System V Release 3.2–STREAMS Primer, AT&T, Prentice Hall, Englewood Cliffs, N.J., 1989.

Unix System V Release 3.2–STREAMS Programmer's Guide, AT&T, Prentice Hall, Englewood Cliffs, N.J., 1989.

Transport Provider Interface Specification, UNIX International, OSI Special Interest Group, Revision 1.5, 1992-12-10.

Network Provider Interface Specification, UNIX International, OSI Work Group, Revision 2.0.0, 1992-08-17.

Slide 7

Character Set Conversions with `iconv`

Problem: texts are witten using different character sets and/or encodings. How to make them readable on all systems?

- Transfer protocols such as HTML specify the content character set in the header.
- ISO C defines the `mbsrtowcs` function family to transform external byte-oriented text to internal wide character text.

  Problem: it uses a global state and conversion between arbitrary character sets is not easily doable.
- XPG2 introduced the `iconv` interface.

Some information on Character Sets

Many different character sets were developed over time, not all of them are still in use. An outdated table of the ISO at

`http://www.isoc.org:8080/codage/registre_ISO.htm`

shows a total of 187 registered character sets. But there are several character sets which are not registered with the ISO so that I would guess the number of character sets in use today at about 100. Many character sets are variants of official versions made by vendors who consider themself above all international collaboration (this include Apple, Microsoft, IBM).

There are several different kinds of character sets and this diversity mackes the transformation process *interesting*.

- 7 bit character sets mostly cover only ISO 646 (often called ANSI), while replacing the one or the other character with a locally used one.
- 8 bit character sets are the used most today for languages which use alphabets. Many are based on ISO 646 by adding new characters in the positions 160 to 255. But there are also character sets which are not based on ISO 646 and nevertheless are in frequent use (e.g., EBCDIC).
- multi-byte character sets, necessary to represent most of the Asian languages with their thousands of glyphs. Many of these character sets are based on the ISO 2022 standard which defines a general framework for these kind of character sets. But there are also completely different approaches.
- wide character sets. There are mainly two sets: Unicode and ISO 10646. Unicode is defined as a subset of ISO 10646.

  The external representation of wide character sets normally is chosen as an encoding which transforms each character in a byte sequence. This is *not* the same as a multi-byte character set.

Slide 8

Character Set Conversions with `iconv`, II

The three defined interfaces are:

```
iconv_t iconv_open(const char *from, const char *to);

size_t iconv(iconv_t handle,
             char **fromptr, size_t *fromsize,
             char **toptr, size_t *tosize);

int iconv_close(iconv_t handle);
```

Using the `iconv` interface

Before diving into the internals of the `iconv` implementation it should be explained how the interface is working. The prototypes already explain this to some extend but nevertheless, an example makes it clearer.

Assume a function which takes a pointer to a buffer and has to write the converted text to a stream. The code for such a function would look like this:

```
int
convert (const char *from, const char *to,
         const char *buf, size_t buflen, FILE *fp)
{
  char buf[32768];
  int result = 0;
  iconv_t handle = iconv_open (to, from);
  if (handle == (iconv_t) -1) return -1;
  while (buflen > 0)
    {
      char *outptr = buf;
      size_t outlen = sizeof (buf);

      size_t n = iconv (handle, &buf, &buflen, &outptr, &outlen);
      if ((n == (size_t) -1 && errno != E2BIG)
          || fwrite (fp, 1, outptr - buf, buf) < (outptr - buf))
        {
          result = -1;
          break;
        }
    }
  iconv_close (handle);
  return result;
}
```

The error handling in this example code is a bit weak but so do not use this piece of code in a production version quality program.

The first two parameters of the function specify the character sets. These are passed to the `iconv_open` function. Please note the order: destination character first. If the system knows about the two character sets and knows how to convert between them it returns a handle for this conversion. This resource has to be freed after use, otherwise the system will sooner or later run out of resources.

The actual conversion function `iconv` takes as the first parameter the handle returned by `iconv_open`. The other parameters specify the input and output buffer since `iconv` converts from one buffer into another. Each buffer is specified by a pointer to a pointer to the buffer and a pointer to a variable containing the length of the buffer. This allows the `iconv` function to update the pointer and length variables which makes the `iconv` function very easy to use as can be seen in the example code.

The only complex thing about `iconv` is the way errors are signaled. There are several different errors. The `glibc` manual and the Unix 98 man pages give more details. At this point we only want to point to two possibly questionable interface details.

As can be seen in the example code above the error code `E2BIG` is returned if the conversion was successful but the output buffer was too small. It is strange that this is signalled as an error and so the programmer has to take care of this case. A case where the output buffer size is limited and multi `iconv` calls are necessary to convert the input buffer is not infrequent.

The second point to mention here is more annoying. If one wants to convert the contents of a file and wants to avoid loading the whole file at once into memory one normally reads the file piecewise and feeds the pieces into the `iconv` function. But if the input text is encoding using a multibyte encoding it is not guaranteed that the last character in the input buffer is complete. The last bytes of a multibyte character can still be left unread. In this case the `iconv` functions stops processing before the incomplete character and sets `errno` to the value `EINVAL`. This is another case which needs special treatment especially since there might still be some bytes left in the input buffer and therefore the input buffer can not immediately be overwritten with newly read text from the input file.

How `iconv` works will become clearer soon as the internals are explained. Here we should have only a few references:

- *The Unicode Standard, Version 2.0*, The Unicode Consortium, 1996. ISBN 0-201-48345-9. The book comes with a CDROM containing conversion and character property tables.

  There is meanwhile a Unix 2.1 standard but no new book is printed. Instead one can get an errata list.

- *ISO/IEC 10646-1:1993 Information technology - Universal Multiple-Octet Coded Character Set (UCS) - Part 1: Architecture and Basic Multilingual Plane.*

  This is the official ISO 10646 standard which costs a lot of money and does not provide any real information which cannot be found anywhere else (e.g., the Unicode standard). It might become in future more interesting as ISO 10646 covers more characters than Unicode, but for now this is not the case.

Slide 9

Character Set Conversions with `iconv`, III

- due to the numerous character sets in use the conversion functions cannot be hardcoded in the C library
- it must be possible to add more conversions once the user needs them

Therefore: conversion are implemented in loadable modules or using tables.

Some existing implementations (in the normal US version):

Solaris 7 216 unique conversion modules.

HP/UX 11.00 about 200 unique, mostly table-based conversions.

Irix 6.3 9 conversions.

Methods used in UNIX `iconv` implementations

For obvious reasons the conversion mechanisms are not built-in to the C library. The two methods to implement this are:

- Using conversion tables which are loaded on demand and processed by one of possibly many generic algorithms for this job.

  The advantage is compact code and data. The disadvantage is that the generic implementation certainly cannot be as efficient as a specific implementation for just that conversion.

- Using conversion functions dynamically loaded into the running program.

  This method can be significantly faster since the functions can be specifically written for the one conversion. The drawback is that a new module for every conversion sums up very quickly. But since generally there are not many conversions needed at one time the needed resources do not grow that much.

All of the implementations mentioned above have one thing in common: the available conversions are all available in a one step process. I.e., if the conversions from $\mathcal{A}$ to $\mathcal{B}$ and from $\mathcal{B}$ to $\mathcal{C}$ are available this does not mean the conversion from $\mathcal{A}$ to $\mathcal{C}$ is also possible in one step. If there is no support for direct conversion from $\mathcal{A}$ to $\mathcal{C}$ one has to convert a text in two steps.

This is a very annoying and hard limitation. The user of the `iconv` function will get as the answer only "no, this conversion is not available". How should it be possible to find out whether these exists a conversion path which might require a number of steps? This is certainly not generally possible.

On some platforms an intelligent guess might be sufficent. Several platforms provide conversions to and from UTF-8 for many character sets. But this is no real solution.

A last problem is that (at least for mere mortals) the specifications of the module interface is not available. So one cannot go on and write the modules which are missing.

Slide 10

`glibc`'s `iconv` implementation

- uses loadable modules which can implement one or more conversions.
- automatically combines available conversion to support the transitive set.
- supports approximately 125 character with more than 23000 pairs.

  SGI: 9, HP/UX, Solaris: 200
- documented interface for the modules to enable third parties writing of new modules.

In the appendix you can find the part of the GNU C library manual covering the `iconv` implementation. It should answer all the questions and show the benefits of this approach.

Slide 11

Upcoming `iconv` development

The `iconv` development is not yet over.

- more supported character sets
- transliteration support (e.g., `"ä"` to `"ae"`)
- transcription support (transcribe, e.g., Cyrillic with Latin alphabet)

Supported character sets in `glibc`

The following list contains the names of the character sets which are supported in the `iconv` implementation in `glibc`. Several names are aliases but these aliases make the implementation really useful as different applications expect different names:

437	500	500V1	850
851	852	855	857
860	861	862	863
864	865	866	869
874	904	1026	1047
8859_1	8859_2	8859_3	8859_4
8859_5	8859_6	8859_7	8859_8
8859_9	10646-1:1993	10646-1:1993/UCS4	ANSI_X3.4-1968
ANSI_X3.4-1986	ANSI_X3.4	ANSI_X3.110-1983	ANSI_X3.110
ARABIC	ARABIC7	ASCII	ASMO-708
ASMO_449	BALTIC	BIG-5	BIG-FIVE
BIG5	BIGFIVE	BS_4730	CA
CN	CP-AR	CP-GR	CP-HU
CP037	CP038	CP273	CP274
CP275	CP278	CP280	CP281
CP284	CP285	CP290	CP297
CP367	CP420	CP423	CP424
CP437	CP500	CP737	CP775
CP819	CP850	CP851	CP852
CP855	CP857	CP860	CP861
CP862	CP863	CP864	CP865
CP866	CP868	CP869	CP870
CP871	CP874	CP875	CP880
CP891	CP903	CP904	CP905
CP918	CP932	CP949	CP1004
CP1026	CP1047	CP1250	CP1251
CP1252	CP1253	CP1254	CP1255
CP1256	CP1257	CP1258	CP1361
CPIBM861	CSA7-1	CSA7-2	CSASCII
CSA_T500-1983	CSA_T500	CSA_Z243.4-1985-1	CSA_Z243.4-1985-2
CSDECMCS	CSEBCDICATDE	CSEBCDICATDEA	CSEBCDICCAFR
CSEBCDICDKNO	CSEBCDICDKNOA	CSEBCDICES	CSEBCDICESA
CSEBCDICESS	CSEBCDICFISE	CSEBCDICFISEA	CSEBCDICFR
CSEBCDICIT	CSEBCDICPT	CSEBCDICUK	CSEBCDICUS
CSEUCKR	CSHPROMAN8	CSIBM037	CSIBM038
CSIBM273	CSIBM274	CSIBM275	CSIBM277
CSIBM278	CSIBM280	CSIBM281	CSIBM284
CSIBM285	CSIBM290	CSIBM297	CSIBM420

CSIBM423	CSIBM424	CSIBM599	CSIBM851
CSIBM855	CSIBM857	CSIBM860	CSIBM863
CSIBM864	CSIBM865	CSIBM866	CSIBM868
CSIBM869	CSIBM870	CSIBM871	CSIBM880
CSIBM891	CSIBM903	CSIBM904	CSIBM905
CSIBM918	CSIBM1026	CSISO4UNITEDKINGDOM	CSISO10SWEDISH
CSISO11SWEDISHFORNAMES	CSISO14JISC6220RO	CSISO15ITALIAN	CSISO16PORTUGESE
CSISO17SPANISH	CSISO18GREEK7OLD	CSISO19LATINGREEK	CSISO21GERMAN
CSISO25FRENCH	CSISO27LATINGREEK1	CSISO49INIS	CSISO50INIS8
CSISO51INISCYRILLIC	CSISO58GB1988	CSISO60DANISHNORWEGIAN	CSISO60NORWEGIAN1
CSISO61NORWEGIAN2	CSISO69FRENCH	CSISO84PORTUGUESE2	CSISO85SPANISH2
CSISO86HUNGARIAN	CSISO88GREEK7	CSISO89ASMO449	CSISO90
CSISO92JISC62991984B	CSISO99NAPLPS	CSISO103T618BIT	CSISO111ECMACYRILLIC
CSISO121CANADIAN1	CSISO122CANADIAN2	CSISO139CSN369103	CSISO141JUSIB1002
CSISO143IECP271	CSISO150	CSISO150GREEKCCITT	CSISO151CUBA
CSISO153GOST1976874	CSISO646DANISH	CSISO2022JP	CSISO2022JP2
CSISO2022KR	CSISO2033	CSISO5427CYRILLIC	CSISO5427CYRILLIC1981
CSISO5428GREEK	CSISO10367BOX	CSISOLATIN1	CSISOLATIN2
CSISOLATIN3	CSISOLATIN4	CSISOLATIN5	CSISOLATIN6
CSISOLATINARABIC	CSISOLATINCYRILLIC	CSISOLATINGREEK	CSISOLATINHEBREW
CSKOI8R	CSKSC5636	CSMACINTOSH	CSNATSDANO
CSNATSSEFI	CSN_369103	CSPC8CODEPAGE437	CSPC8LATINHEBREW
CSPC8MULTILINGUAL	CSPC775BALTIC	CSPCP852	CSSHIFTJIS
CUBA	CWI-2	CWI	CYRILLIC
DE	DEC-MCS	DEC	DIN_66003
DK	DS2089	DS_2089	E13B
EBCDIC-AT-DE-A	EBCDIC-AT-DE	EBCDIC-BE	EBCDIC-BR
EBCDIC-CA-FR	EBCDIC-CP-AR1	EBCDIC-CP-AR2	EBCDIC-CP-BE
EBCDIC-CP-CA	EBCDIC-CP-CH	EBCDIC-CP-DK	EBCDIC-CP-ES
EBCDIC-CP-FI	EBCDIC-CP-FR	EBCDIC-CP-GB	EBCDIC-CP-GR
EBCDIC-CP-HE	EBCDIC-CP-IS	EBCDIC-CP-IT	EBCDIC-CP-NL
EBCDIC-CP-NO	EBCDIC-CP-ROECE	EBCDIC-CP-SE	EBCDIC-CP-TR
EBCDIC-CP-US	EBCDIC-CP-WT	EBCDIC-CP-YU	EBCDIC-CYRILLIC
EBCDIC-DK-NO-A	EBCDIC-DK-NO	EBCDIC-ES-A	EBCDIC-ES-S
EBCDIC-ES	EBCDIC-FI-SE-A	EBCDIC-FI-SE	EBCDIC-FR
EBCDIC-GREEK	EBCDIC-INT	EBCDIC-INT1	EBCDIC-IS-FRISS
EBCDIC-IT	EBCDIC-JP-E	EBCDIC-JP-KANA	EBCDIC-PT
EBCDIC-UK	EBCDIC-US	ECMA-114	ECMA-118
ECMA-CYRILLIC	ELOT_928	ES	ES2
EUC-CN	EUC-JP	EUC-KR	EUC-TW
EUCCN	EUCJP	EUCKR	EUCTW
FI	FR	GB	GB_1988-80
GOST_19768-74	GOST_19768	GREEK-CCITT	GREEK
GREEK7-OLD	GREEK7	GREEK8	HEBREW
HP-ROMAN8	HU	IBM037	IBM038
IBM256	IBM273	IBM274	IBM275
IBM277	IBM278	IBM280	IBM281
IBM284	IBM285	IBM290	IBM297
IBM367	IBM420	IBM423	IBM424
IBM437	IBM500	IBM775	IBM819
IBM850	IBM851	IBM852	IBM855
IBM857	IBM860	IBM861	IBM862
IBM863	IBM864	IBM865	IBM866
IBM868	IBM869	IBM870	IBM871
IBM875	IBM880	IBM891	IBM903
IBM904	IBM905	IBM918	IBM1004
IBM1026	IBM1047	IEC_P27-1	INIS-8
INIS-CYRILLIC	INIS	ISO-2022-JP-2	ISO-2022-JP
ISO-2022-KR	ISO-8859-1	ISO-8859-2	ISO-8859-3
ISO-8859-4	ISO-8859-5	ISO-8859-6	ISO-8859-7
ISO-8859-8	ISO-8859-9	ISO-8859-10	ISO-8859-11
ISO-8859-13	ISO-8859-14	ISO-8859-15	ISO-10646
ISO-10646/UCS2	ISO-10646/UCS4	ISO-10646/UTF-?8	ISO-10646/UTF8
ISO-IR-4	ISO-IR-6	ISO-IR-8-1	ISO-IR-9-1
ISO-IR-10	ISO-IR-11	ISO-IR-14	ISO-IR-15
ISO-IR-16	ISO-IR-17	ISO-IR-18	ISO-IR-19
ISO-IR-21	ISO-IR-25	ISO-IR-27	ISO-IR-37
ISO-IR-49	ISO-IR-50	ISO-IR-51	ISO-IR-54
ISO-IR-55	ISO-IR-57	ISO-IR-60	ISO-IR-61
ISO-IR-69	ISO-IR-84	ISO-IR-85	ISO-IR-86
ISO-IR-88	ISO-IR-89	ISO-IR-90	ISO-IR-92
ISO-IR-98	ISO-IR-99	ISO-IR-100	ISO-IR-101
ISO-IR-103	ISO-IR-109	ISO-IR-110	ISO-IR-111
ISO-IR-121	ISO-IR-122	ISO-IR-126	ISO-IR-127
ISO-IR-138	ISO-IR-139	ISO-IR-141	ISO-IR-143
ISO-IR-144	ISO-IR-148	ISO-IR-150	ISO-IR-151
ISO-IR-153	ISO-IR-155	ISO-IR-156	ISO-IR-157
ISO-IR-179	ISO646-CA	ISO646-CA2	ISO646-CN

ISO646-CU	ISO646-DE	ISO646-DK	ISO646-ES
ISO646-ES2	ISO646-FI	ISO646-FR	ISO646-FR1
ISO646-GB	ISO646-HU	ISO646-IT	ISO646-JP-OCR-B
ISO646-JP	ISO646-KR	ISO646-NO	ISO646-NO2
ISO646-PT	ISO646-PT2	ISO646-SE	ISO646-SE2
ISO646-US	ISO646-YU	ISO6937	ISO_646.IRV:1991
ISO_2033-1983	ISO_2033	ISO_5427-EXT	ISO_5427
ISO_5427:1981	ISO_5428	ISO_5428:1980	ISO_6937-2
ISO_6937-2:1983	ISO_6937	ISO_6937:1992	ISO_8859-1
ISO_8859-1:1987	ISO_8859-2	ISO_8859-2:1987	ISO_8859-3
ISO_8859-3:1988	ISO_8859-4	ISO_8859-4:1988	ISO_8859-5
ISO_8859-5:1988	ISO_8859-6	ISO_8859-6:1987	ISO_8859-7
ISO_8859-7:1987	ISO_8859-8	ISO_8859-8:1988	ISO_8859-9
ISO_8859-9:1989	ISO_8859-10	ISO_8859-10:1993	ISO_8859-14:1998
ISO_8859-15:1998	ISO_9036	ISO_10367-BOX	IT
JIS_C6220-1969-RO	JIS_C6229-1984-B	JOHAB	JP-OCR-B
JP	JS	JUS_I.B1.002	KOI-7
KOI-8	KOI8-R	KSC5636	L1
L2	L3	L4	L5
L6	L7	L8	LATIN-GREEK-1
LATIN-GREEK	LATIN1	LATIN2	LATIN3
LATIN4	LATIN5	LATIN6	LATIN7
LATIN8	MAC-IS	MAC-UK	MAC
MACINTOSH	MS-ANSI	MS-ARAB	MS-CYRL
MS-EE	MS-GREEK	MS-TURK	MSCP949
MSCP1361	MSZ_7795.3	MS_KANJI	NAPLPS
NATS-DANO	NATS-SEFI	NC_NC00-10	NC_NC00-10:81
NF_Z_62-010	NF_Z_62-010_(1973)	NF_Z_62-010_1973	NO
NO2	NS_4551-1	NS_4551-2	OS2LATIN1
OSF00010001	OSF00010002	OSF00010003	OSF00010004
OSF00010005	OSF00010006	OSF00010007	OSF00010008
OSF00010009	OSF0001000A	OSF00010020	OSF00010100
OSF00010101	OSF00010102	OSF00010104	OSF00010105
OSF00010106	OSF00030010	OSF0004000a	OSF0005000a
OSF05010001	OSF100201A4	OSF100201A8	OSF100201B5
OSF100201F4	OSF100203B5	OSF1002011C	OSF1002011D
OSF1002035D	OSF1002035E	OSF1002035F	OSF1002036B
OSF1002037B	OSF10010001	OSF10020025	OSF10020111
OSF10020115	OSF10020116	OSF10020118	OSF10020122
OSF10020129	OSF10020352	OSF10020354	OSF10020357
OSF10020359	OSF10020360	OSF10020364	OSF10020365
OSF10020366	OSF10020367	OSF10020370	OSF10020387
OSF10020388	OSF10020396	OSF10020402	OSF10020417
PT	PT2	R8	ROMAN8
SE	SE2	SEN_850200_B	SEN_850200_C
SHIFT-JIS	SJIS	SS636127	ST_SEV_358-88
T.61-8BIT	T.61	UCS-2	UCS-4
UCS2	UCS4	UHC	UK
UNICODE	UNICODEBIG	UNICODELITTLE	US-ASCII
US	UTF-8	UTF8	WINBALTRIM
YU			

Slide 12

Simplified Signal Handling

Coming from the early System V standard comes yet another signal handling interface.

`int sighold (int sig)` – Add SIG to the calling process' signal mask.

`int sigrelse (int sig)` – Remove SIG from the calling process' signal mask.

`int sigignore (int sig)` – Set the disposition of SIG to SIG_IGN.

`__sighandler_t sigset (int sig, __sighandler_t disp)` – Set the disposition of SIG.

This signal handling interface really should not be used. It suffers from the same problems as the `signal` interface defined in ISO C. Using this interface it is not possible to fine-tune the signal behaviour.

Every new program should use the signal handling functions defined in POSIX.1. They are widely portable, they let the programmer determine every aspect of the program behaviour when it comes to interaction with signals and signal handlers.

Slide 13

System V Pseudo Terminals

Linux initially had support for the BSD-style pseudo terminals.

Linux 2.2 introduces support for System V style pseudo-terminals:

1. `open` master pseudo terminal
2. call `grantpt` with the retrieved descriptor
3. unlocked the slave device with `unlockpt`
4. get the slave device name with `ptsname`
5. open the slave device

The `devfs` filesystem automatically generates the slave pseude devices. The hundreds of master/slave pairs are not necessary anymore.

How to use the System V PTYs

This should not be a tutorial on programming. A complete handling of the problems of pseudo TTY handling can be found in Unix Network Programming by Richard Stevens [UnixNet]. Section 15.4 handles the problem in detail, for System V and BSD style PTYs.

The most interesting of the new functions is `grantpt`. The task of this function is it to allow the program to actually use the slave TTY by changing the ownership. The kernel creates the slave TTY completely protected.

To perform the mode changing an external program named `pt_chown` is used. This is the way most System V derived systems implement this. The `grantpt` function forks, and executes the program. The `pt_chown` program is a SUID root binary. It therefore can change the access permissions. Once the `pt_chown` program terminates the `grantpt` function returns and the pseude terminal is usable for the current user.

The first releases of the Linux distribution which use `glibc 2.1` will use the System V pseudo terminal handling at least for some key-programs like `telnet`. All programs which use the function `openpty` defined in the C library also will automatically take advantage of the new mechanism.

The extra program `pt_chown` used by `grantpt` could be an extra source of problems. Therefore it is necessary that this program is always in its correct place (normally this is `/usr/sbin/pt_chown`) and of course that it is correctly compiled. In case the system does not allow logins one should first check whether this program works correctly.

Slide 14

Large File Support

The X/Open Group published a proposal to support files with length $> 2^{31}$ bytes. `glibc 2.1` has complete support for this proposal as far as the kernel allows it.

- Function and data structures dealing with file size or offsets have counterparts named by adding `64` to the names (e.g., `fseek64`)
- When using a special compilation mode the LFS interface completely takes over and transparently replaces to 32 bits interface.

An older version of the LFS (Large File Support) specification is available at

`http://www.sas.com/standards/large.file/x_open.20Mar96.txt`

The implementation in `glibc 2.1` follows the most recent draft of the specification published in the *Go Solo 2* book by the X/Open Group [GoSolo2]. Chapter 16 describes the externsion.

The changed and extended functions are:

`aio_read`	`aio_write`	`exec`	`fcntl`	`fdopen`	`fgetpos`	`fgetc`
`fgets`	`fread`	`fscanf`	`getc`	`getchar`	`gets`	`getw`
`scanf`	`fopen`	`freopen`	`tmpfile`	`fpathconf`	`pathconf`	`fprintf`
`fputs`	`fwrite`	`printf`	`putc`	`putchar`	`puts`	`putw`
`vfprintf`	`vprintf`	`fseek`	`fseeko`	`fstat`	`lstat`	`stat`
`fstatvfs`	`ststvfs`	`ftello`	`ftruncate`	`getrlimit`	`setrlimit`	`lio_listio`
`lseek`	`mmap`	`open`	`read`	`readv`	`readdir`	`write`
`writev`						

How to use the LFS functionality

There are two basic ways to use the LFS:

1. The source code can be rewritten (or written like this right from the beginning) to use the LFS counterparts of the functions which changed. E.g., uses of `lseek` would be replaced by `lseek64` and the type `off_t` would be replaced by `off64_t`.

   To compile such a changed program the macro `_LARGEFILE64_SOURCE` must be compiled before the first system header file is included.

```
cc -D_LARGEFILE64_SOURCE         \
   `getconf LFS64_CFLAGS` source.c \
   `getconf LFS64_LDFLAGS`         \
   `getconf LFS64_LIBS`
```

The use of `getconf` ensures that all necessary flags and libraries are used.

2. The program can throughout use the large file support. No code changes are necessary to do this. To enable this mode the compiler should be called like this:

```
cc -D_FILE_OFFSET_BITS=64        \
   `getconf LFS_CFLAGS` source.c \
   `getconf LFS_LDFLAGS`         \
   `getconf LFS_LIBS`
```

Latest versions of the GNU utilities (e.g., `textutils`) support LFS. They also contain tests for LFS which could be copied in own projects.

Current limitations

Since the Linux kernel in general does not support large files on 32 bits systems in the moment the usability is restricted. The only function supported is `lseek`, with the `lseek64` equivalent, the stream I/O counterparts (`fseek` and `fseek64`) and `open` (which has not to be changed on Linux).

The benefits of the `glibc` changes are that one can now write program using the new interface which in the moment are limited by the filesize limit. But the same program, without the need to be recompiled, can be used once the kernel is extended appropriately. This is expected to be happening in the early phase of the Linux 2.3 development.

Slide 15

The `fmtmsg` family of functions

System tools on System V systems print their messages in a uniform way and the user can select the level of verbosity.

```
int fmtmsg (long int classification, const char *label,
            int severity, const char *text,
            const char *action, const char *tag)
```

Example output:

```
UX:cat: NOTE2: invalid syntax
TO FIX: refer to manual UX:cat:001
```

Environment variables `MSGVERB` and `SEV_LEVEL` control the output.

Using `fmtmsg`?

Somebody who worked with System V system certainly will appreciate the uniformity of the messages printed by the various system utilities. This makes it easy for the novice to find her/his way. The messages the tools emit even help to locate the appropriate place in a manual where the reason for the message is described.

The uniformity is no surprise since all the tools use the `fmtmsg` function which enforces this. The other nice thing about this function is that it lets the user at runtime decide what level of information s/he wants. The advanced user certainly does not want to see the references to the manual all the time since s/he knows these details.

There are a few problem with using this function, though.

- The function itself is inflexible. It expects strings as parameters which then are simply copied in the output.

  On the other hand the `error` function defined in `glibc` allows the user to specify a format string and arbitrary many parameters. This allows to move the code to generate the message text in the library function and does not force the user to code this over and over again.

- The look&feel currently used on GNU systems is very different. The GNU coding standard prescribes the format of warning and error messages and this format is very different from the System V style.

  It is therefore questionable whether it is worth it to make such a break in the user interface.

The `glibc` manual provides complete documentation of the functions, together with example programs. Get it through

```
info 'libc' 'Printing Formatted Messages'
```

Slide 16

New POSIX interfaces

With the improving Linux kernel it was possible to implement several more of the POSIX interfaces.

- `wordexp` is implemented (last missing POSIX.2 interface)
- POSIX real-time signals
- POSIX asynchronous I/O
- read-write locks in thread library

Compliance Statement

One of the major goals of the Linux development is it to be compliant with the POSIX standards (wherever this is reasonable). The standards in question are:

- POSIX.1: basic system interfaces
- POSIX.1b: real-time programming interface
- POSIX.1c: thread interface
- POSIX.2: system utilities

`glibc 2.1` should be fully compliant with the POSIX.2 standard when it comes to the library functions and the utilities belonging to them. Most of the standard deals with the system utilities like `ls` and these are not part of `glibc`. (Nevertheless the GNU versions of these programs are expected to be POSIX complied.)

The compliance with the various POSIX.1 standards is determine by ther kernel. Almost all the functionality requires help from the kernel and if this help cannot be provided the system is not compliant.

The good news is that almost everything works. There are only a few things missing:

- `clone` has no possibility to start threads suspended. This makes it impossible to implement certain aspects of the thread startup correctly (e.g., setting the priority).
- the signal handling in multi-threaded programs cannot be implemented correctly. This is closely related to the next point.
- all created threads have different process IDs.
- global POSIX semaphores, shared memory segments and message queues are not implemented.

Beside this there should not be any further compliance problems. It the Linux kernel finally gets support for these missing features the `glibc` implementation will follow immediately and the dream of a completely POSIX compliant system becomes true.

Slide 17

POSIX Real-time signals

- Introduced early in the Linux 2.1 development
- real-time signals allow
  - to queue signals. Signals arriving which there already is such a signal waiting for the process to accept it are not automatically lost.
  - signals can be sent together with data
  - provide a nicer interface as `sigcontext` to access extra information.
- provide more signals the program can use (not only `SIGUSR1` and `SIGUSR2`). This was critical for the thread implementation.

Real-time signals are those in the range `SIGRTMIN` to `SIGRTMAX`. The POSIX standard explicitly says that these symbols can be macros which do not evaluate to compile-time constant values.

This is used in the `glibc` implementation to allow real-time signals to be used in the implementation itself. More specific: the implementation of the currently used thread library depends on signals. It formerly used `SIGUSR1` and `SIGUSR2` which lead to problems because programs which use threads sometimes also want to use these signals. Beside, we need a third signal for the debugger interface.

For this reason the `glibc` implements a machanism which allows the implementation to allocate real-time signals before the user-code is started. It then completely hides these allocated signals so that programs which follow the POSIX standard and do not assume a compile time constant `SIGRTMIN` value don't even notice the "stolen" signals. (There is a `gdb` version supporting thread, official support is also underway.)

This is signal allocation is implemented using a function to actually allocate the signal and another function to query the first available real-time signal. This function is then used in the `SIGRTMIN` and a variant in the `SIGRTMAX` macro. The actual definition looks like this:

```
#define SIGRTMIN (__libc_current_sigrtmin ())
```

The other extremely useful extension coming with the real-time signal implementation is the possibility to partially use the new techniques with the traditional signals. Those signals will never queue up but using the `SA_SIGINFO` flag it is possible to specify with `sigaction` that the kernel should send a reference to an object of type `siginfo_t` together with the signal. The consequences of this is that, e.g., program which have to find the address where a segmention fault happened, now can be written in a portable way.

Before the signal handler had to handle the system and architecture specific `sigcontext` structure. In these objects, somewhere was the address. Using the `siginfo_t` objects the address is available in an officially documented way. At least this is the theory, the kernel still lacks some of the functionality.

Example code to use the `siginfo_t` structure:

```
void
handler (int sig, siginfo_t *info, void *data)
{
  printf ("segv at %x\n", info->si_addr);
  _exit (1);
}

{
  ...
  sa.sa_sigaction = handler;
  memset (&sa.sa_mask, '\0', sizeof (sigset_t));
  sa.sa_flags = SA_SIGINFO;
  sigaction (SIGSEGV, &sa, NULL);

}
```

As one can see the function catching the signal is trivial now. The name of the structure element `si_addr` is standardized and therefore porting this program will make no big problems.

Slide 18

POSIX Asynchronous I/O

- demanded by database, multimedia, and scientific software writers
- replaces less powerful, non-portable existing implementations
- can be implemented on user- or kernel-level
- current `glibc` implementation is completely user-level based

Using AIO

There are so far not many examples on how to use POSIX asynchronous I/O support. The POSIX standard specifies all the functions but this is not a big help since there are many possible ways to use these functions together and the standard only describes the functions and their behaviour.

The best introduction I know comes from [Gall]. Chapter 6 of the book covers I/O problems "for the real world". The book is a bit old meanwhile but has not many contesters.

The interfaces are defined in the file `aio.h` and the functions are available in the library `librt`. They are not in the `libc` object themself. The name `librt` is prescribed by Unix98. Many other systems have a `libaio`, so one has to be prepared for portability problems.

```
struct aiocb
{
  int aio_fildes; /* File desriptor.  */
  int aio_lio_opcode; /* Operation to be performed.  */
  int aio_reqprio; /* Request priority offset.  */
  volatile void *aio_buf; /* Location of buffer.  */
  size_t aio_nbytes; /* Length of transfer.  */
  struct sigevent aio_sigevent; /* Signal number and value.  */
  ...
}
```

These are the elements of the control structure used by all operations. The user has to fill in appropriate values before calling one of the follwing functions:

```
/* Enqueue read request for given number of bytes and the given priority.  */
extern int aio_read (struct aiocb *aiocbp);
```

```
/* Enqueue write request for given number of bytes and the given priority.  */
extern int aio_write (struct aiocb *aiocbp);

/* Initiate list of I/O requests.  */
extern int lio_listio (int mode,
                       struct aiocb *const list[],
                       int nent,
                       struct sigevent *restrict sig);
```

These are the basic I/O functions. The first two are obviously for reading and writing. The third function can process a whole bunch of requests at once. The `list` array specifies all the actions which have to be performed, the `mode` argument how the caller behaves (waits, gets no notification, gets a signal).

Beside this there are functions for synchronization, to cancel requests and to query the error status. For all I/O functions there also is a LFS counterpart, as defined in the LFS proposal.

Slide 19

Read-Write Locks

- POSIX defines only mutexes which allow exactly one user (e.g., database updates)
- in many situations a program very well can execute many read operations but only one write operations

Solution: Read-Write locks.

→ they allow arbitrary many readers, but only on writer no reader at the same time.

→ prioritizing (writer or reader preferred) selectable

This extension to the POSIX interface is defined in Unix98. The definitions are very close to the POSIX thread mutex definitions, with the obviously necessary extensions:

```
/* Static initializer.  */
#define PTHREAD_RWLOCK_INITIALIZER ...

/* Initialize read-write lock RWLOCK using attributes ATTR, or use
   the default values if later is NULL.  */
extern int pthread_rwlock_init (pthread_rwlock_t *rwlock,
const pthread_rwlockattr_t *attr);

/* Destroy read-write lock RWLOCK.  */
extern int pthread_rwlock_destroy (pthread_rwlock_t *rwlock);

/* Acquire read lock for RWLOCK.  */
extern int pthread_rwlock_rdlock (pthread_rwlock_t *rwlock);

/* Try to acquire read lock for RWLOCK.  */
extern int pthread_rwlock_tryrdlock (pthread_rwlock_t *rwlockd));

/* Acquire write lock for RWLOCK.  */
extern int pthread_rwlock_wrlock (pthread_rwlock_t *rwlock);

/* Try to acquire writelock for RWLOCK.  */
extern int pthread_rwlock_trywrlock (pthread_rwlock_t *rwlock);

/* Unlock RWLOCK.  */
extern int pthread_rwlock_unlock (pthread_rwlock_t *rwlock);
```

Slide 20

ISO C 9x news

- ISO C 9x is a *major* revision of the ISO C89 standard
- more than 140 new interfaces. Main groups:
  - new complex and imaginary data types
  - math library
  - math exception handling
  - extended integer types
  - `printf`/`scanf` extensions
  - time function extensions

A draft of the standard is at the time this was written available at

`http://anubis.dkuug.dk/JTC1/SC22/WG14/www/docs/n869.ps.gz`

Due to the policies of the ISO it is unfortunately not possible to the make the standard freely available for everybody. Therefore it could be that this link already is stale.

Comments on the standard might still be useful. To take influence it is necessary to contact the national standard body which represents the own country in the committee (e.g., ANSI for the US, DIN for Germany). Only paying members have the right to vote but it might be worthwhile trying to contact a person in the committee and ask them to bring the points up.

The biggest group of changes are in the math library. The great work of the NCEG working group (the Numerical C Extensions Group) finally found its way in the standard. Due to this is C one step closer to the goal to replace FORTRAN as the major language in the scientific world.

Slide 21

New Math Interfaces

Several groups of new functions:

- `float` and `long double` versions of existing functions
- more real-value, rounding, and conversion functions
- complex value variants of many functions
- type-generic functions
- math exception handling functions

The number of new interfaces is too large. Therefore the following text only describes a few of the more interesting interfaces. The complete list of changed or new math interfaces relative to `glibc 2.0` is following. See the `glibc` manual for documentation.

cacos	cacosf	cacosh	cacoshf	cacoshl
cacosl	carg	cargf	cargl	casin
casinf	casinh	casinhf	casinhl	casinl
catan	catanf	catanh	catanhf	catanhl
catanl	ccos	ccosf	ccosh	ccoshf
ccoshl	ccosl	cexp	cexpf	cexpl
cimag	cimagf	cimagl	clog	clog10
clog10f	clog10l	clogf	clogl	conj
conjf	conjl	cpow	cpowf	cpowl
cproj	cprojf	cprojl	creal	crealf
creall	csin	csinf	csinh	csinhf
csinhl	csinl	csqrt	csqrtf	csqrtl
ctan	ctanf	ctanh	ctanhf	ctanhl
ctanl	exp10	exp10f	exp10l	exp2
exp2f	exp2l	fdim	fdimf	fdiml
feclearexcept	fegetenv	fegetexceptflag	fegetround	feholdexcept
feraiseexcept	fesetenv	fesetexceptflag	fesetround	fetestexcept
feupdateenv	fma	fmaf	fmal	fmax
fmaxf	fmaxl	fmin	fminf	fminl
llrint	llrintf	llrintl	llround	llroundf
llroundl	log2	log2f	log2l	lrint
lrintf	lrintl	lround	lroundf	lroundl
nan	nanf	nanl	nearbyint	nearbyintf
nearbyintl	nextafterx	nextafterxf	remquo	remquof
remquol	round	roundf	roundl	scalbln
scalblnf	scalblnl	tgamma	tgammaf	tgammal
trunc	truncf	truncl		

Slide 22

Some new real-value functions

`double nearbyint (double x)` – round without raising exception

`double rint (double x)` – round using current direction

`double round (double x)` – round away from zero

`long int lrint (double x)` – round to `long int`

`long long int llrint (double x)` – round to `long long int`

`long int lround (double x)` – round to `long int`

`long long int llround (double x)` – round to `long long int`

`double fma (double x, double y, double z)` – Multiply-Add

Rounding

Accurate rounding was a problem in ISO C89. The IEEE 754 standard specifies many nice features such as rounding modes but there were no standardized C functions to use this.

The extended math libraries coming with System V and some BSD systems included many additional functions on top of the set specified in ISO C89. `glibc 2.0` already had these functions but there still was no guarantee for portability.

Somebody's pet

The `fma` function (floating-point muliply and add) is a very special beast. It multiplies the first two arguments and then adds the third. There never was a good reason to have such a function until some FPUs got an operation which performs exactly this action.

It is not possible to use this operation directly in compilers which easily could recognize constructs like `(a * b) + c`. The problem is that it does not always lead to the same results when not using the new operation since intermediate results are rounded. Therefore some people urged the committee to add this new function.

Slide 23

More new real-value functions

`double fdim (double x, double y)`

– positive difference between `x` and `y`

`double fmax (double x, double y)` – maximum of `x` and `y`

`double fmin (double x, double y)` – minimum of `x` and `y`

`int fpclassify (`*type* `x)` – classify FP number

`int signbit (`*type* `x)` – state of sign bit

`int isfinite (`*type* `x)` – test for finite number

`int isnormal (`*type* `x)` – test for normal number

`int isnan (`*type* `x)` – test for NaN

`int isinf (`*type* `x)` – test for Infinity

Portability aspects

The need for functions like `fpclassify` and `isfinite` is not new. All Unix systems provide implementations for the one or the other function. As an example, this is what Sun provides in Solaris 7:

```
typedef enum fpclass_t {
        FP_SNAN = 0,    /* signaling NaN */
        FP_QNAN = 1,    /* quiet NaN */
        FP_NINF = 2,    /* negative infinity */
        FP_PINF = 3,    /* positive infinity */
        FP_NDENORM = 4, /* negative denormalized non-zero */
        FP_PDENORM = 5, /* positive denormalized non-zero */
        FP_NZERO = 6,   /* -0.0 */
        FP_PZERO = 7,   /* +0.0 */
        FP_NNORM = 8,   /* negative normalized non-zero */
        FP_PNORM = 9    /* positive normalized non-zero */
} fpclass_t;

extern fpclass_t fpclass(double);       /* get class of double value */
extern int finite(double);
extern int unordered(double, double);
```

But there is a major difference to the ISO C 9x functions: all systems provide (so far) only real functions. The ISO C 9x additions are in fact *macros*. The benefit is that one can use the same function for all the different floating-point datatypes supported by the architecture.

It is therefore necessary to understand the portability issues. One cannot rely on the availability of any of the new functions on all systems and the possible replacements have often very different names and interfaces and are often only available for `double`.

Slide 24

Complex math functions

`complex` and `imaginary` type modifiers:

```
complex double a = 2.0 + 3.4 * I;
imaginary double b = 4.0;
```

Accessing via:

`double creal (complex double x)`

`double cimag (complex double x)`

Almost complex set of functions to complement the real-value functions.

GCC and `complex`

GCC supports complex numbers already for quite some time. Since `complex` was not standardized and the symbol would be in the global name space it is called `__complex`.

There is also a method to access the real and imaginary parts:

```
complex double a;
__real__ a = 3.0;
__imag__ a = 4.0;
```

But this should not be used anymore if the `creal` and `cimag` functions are available. The compiler should be able to optimize expressions like

```
complex double a = creal (b) + 4.0 * I;
```

See

```
info gcc 'C Extensions' Complex
```

for more information on GCC's complex number implementation.

Slide 25

Complex Math Functions

All but the most complex math library functions have complex counterparts:

```
complex double csin (complex double x);

complex double clog (complex double x);

double cabs (complex double x);

double cpow (complex double x, complex double y);
```

Possible Conflicts

Many of the additions of the ISO C9x standard possibly introduce name conflicts. The functions are in the global namespace but were not resserved in ISO C89 for future use.

There are two actual problems known:

`clog` This functino obviously implements the complex natural logarithms. The choice of the name is obvious and every other choice would have introduced problems.

The problem is that the C++ language uses this symbol for a standard stream in the I/O stream implementation. This is a global variable and certainly cannot simply be renamed.

In the GNU C++ library the problem is solved using some `gcc` magic. References to the variable `clog` automatically changed into a reference of the variable `__IO_clog`. So the problem is avoided very early in the process.

`cabs` The situation for `cabs` is a bit different. Earlier Unix systems already provided a function with this name. The functionality also was the same. The only difference was: the old implementation expected two `double` arguments while the ISO C9x version expects one `complex double` argument. The effect might be the same for some platform but is could lead to problems on others.

There is no good solution for these problems. Both functions are really useful and shouldn't be avoided in general.

Determined Boundary Cases

The NCEG does not only came up with a large set of new interfaces. It also provided an accurate description of the behaviour of the implementation beyond what mathematics determines.

This description of the function is based on the assumption that IEEE 754 compatible floating-point arithmetic is used. Floating-point implementations following this standard provide beside normal numbers also two special values:

Infinity This special value, which can possibly be signed, represents the result of a computation which would have a definite value but is beyond the representable range of numbers for the given floating-point format.

NaN This is the abbreviation for "Not a Number". This value is returned as the result in case there is no definite result. The IEEE 754 distinguishes between signalling and non-signalling NaNs but the ISO C9x standard does not mentioned signalling NaNs.

A second extension is that the number zero can have a sign. This was introduced to preserve as much information as possible. The example below contains examples of the use of a signed zero value.

And a third are floating-point exceptions. The FPU (or the software emulation) is supposed to raise certain exception in case of specified conditions during the computations. The ISC 9x standard also finally specifies when and how the library function have to handle exceptions.

An example for the way the ISO C9x standard defines the functions is given below. This is the description of the `cexp` function which is one of the fundamental functions used in the implementation of several others:

- `cexp(conj(`z`))` = `conj(cexp(`z`))`.
- `cexp(`$\pm 0 + i0$`)` returns $1 + i0$.
- `cexp(`$+\infty + i0$`)` returns $+\infty + i0$.
- `cexp(`$-\infty + i\infty$`)` returns $\pm 0 \pm i0$ (where the signs of the real and imaginary parts of the result are unspecified).
- `cexp(`$+\infty + i\infty$`)` return $\pm\infty + i\mathrm{NaN}$ and raises the *invalid* exception (where the sign of the real part of the result is unspecified).
- `cexp(`$x + i\infty$`)` return $\mathrm{NaN} + i\mathrm{NaN}$ and raises the invalid exception, for finite x.
- `cexp(`$-\infty + iy$`)` returns $+0(\cos y + i \sin y)$, for finite y.
- `cexp(`$+\infty + iy$`)` returns $+\infty(\cos y + i \sin y)$, for nonzero y.
- `cexp(`$-\infty + i\mathrm{NaN}$`)` returns $\pm 0 \pm i0$ (where the signs of the real and imaginary parts of the result are unspecified).
- `cexp(`$+\infty + i\mathrm{NaN}$`)` returns $\pm\infty + i\mathrm{NaN}$ (where the ign of the real part of the result is unspecified).
- `cexp(`$\mathrm{NaN} + i0$`)` returns $\mathrm{NaN} + i0$.
- `cexp(`$\mathrm{NaN} + iy$`)` returns $\mathrm{NaN} + i\mathrm{NaN}$ and optionally raises the *invalid* exception, for all non-zero numbers y.
- `cexp(`$x + i\mathrm{NaN}$`)` returns $\mathrm{NaN} + i\mathrm{NaN}$ and optionally raises the *invalid* exception for finite x.

Slide 26

Type-Generic Math Macros

- ISO C 9x defines for each type `float`, `double`, and `long double` an own version of a function.
- ISO C++ uses templates

$\Longrightarrow$ ISO C committee wants to have similar mechanisms using the preprocessor

Not implementable using the standardized ISO C functionality.

`gcc` can implement it using `typeof`.

Jealousy

One of the goals for ISO C 9x was to make the language usable and attractive for numeric programming. The library was significantly extended and there are now variants of all functions for all three floating-point types.

But this is exactly the problem. The programmer has to carefully select the function to call. Reusing the same source code for the implementations using different data types is not easy. A frequent case is where a header defines some more complex functions using the available functions of the math library.

```
#define tanh(x) ((exp (x) - exp (-x)) / (exp (x) + exp (-x)))
```

(The above example is not useful in the real-world.) The example code works just fine for `double` but it is not what is wanted for `float` and `long double`. In the former case the `exp` function calls are unnecessarily slow. We are only interested in the precision of the result which up to the precision of `float`. For `long double` the problem is that when calling `exp` the `x` is rounded to a `double` value and we loose precision. The only alternative would be to write three macros.

With the introduction of the macros from the `tgmath.h` header it is now possible to use above code unchanged to use it with all three floating-point types. The only necessary change to to include the `tgmath.h` file.

Those who are interested and fascinated by C preprocessor magix might want to take a look at the definitions of the macros in `glibc`'s version of `tgmath.h`. For the other: don't bother, it's ugly. Just one thing should be know to everybody using the functionality from this header:

Never compile the files without optimization.

Otherwise you'll be surprised about the size of the binaries.

Slide 27

Math Exception Handling

Selecting rounding modes, masking IEEE floating-point interrupts was previously not possible in a portable way:

- `void feclearexcept(int except)`
- `void fegetexceptflag(fexcept_t *flagp, int excepts)`
- `void feraiseexcept(int excepts)`
- `void fesetexceptflag(const fexcept_t *flagp, int exc)`
- `int fetestexcept(int excepts)`
- `int fegetround(int rounding_direction)`
- `int fesetround(int rounding_direction)`
- `void fegetenv(fenv_t *envp)`
- `int feholdexcept(fenv_t *envp)`
- `int feholdexcept(fenv_t *envp)`
- `void fesetenv(const fenv_t *envp)`
- `void feupdateenv(const fenv_t *envp)`

IEEE 754 Exceptions in C

IEEE 754 is the standard of all of today's floating-point implementations follow. Exceptions are rare (Vax, Cray except the Alpha based products). One key feature of this standard is the ability to signal various cases where the computation does not lead to a reasonable result by exceptions.

These mechanisms are already used in numeric programming since the first FPU implementations supporting it but there was no standardized support in the C libraries available. ISO C 9x partly fills the holes. It defines functions to temporarily disable the generation of signal for exceptions and functions to modify the exception bits in the status word.

The introduction of these functions does not require any changes in existing programs. They continue to run. But newly written code using floating-point computations should take them into account. There are basically three categories:

- The program does not expect any exceptions. All computations have representable results. In this case none of the new functions should be used. At least on systems using `glibc` the default behaviour is to generate `SIGFPE` exceptions. So it is easy to see where the program violates the assumptions.
- If the program is known to raise exceptions but one does not care about them. In this situation it is best to use

  feholdexcept (&env);

  to disable generating symbols. This will be as portable as possible.
- Programs which are aware of exceptions and must handle them appropriately have to be written a bit more careful. The parts of the sources where exceptions can be raised and where it is necessary to handle them should be written similar to this:

```
fenv_t env;
feholdexcept (\&env);
```

```
... /* Here comes the expression.  */
if (fetestexcept (FE_UNDERFLOW))
  ... /* Handle underflow.  */
/* Similar action for the other interesting exceptions.  */
...
fesetenv (\&env);
```

What happens is that for the time of the executiong of the critical code the generation of exceptions is disable. Nevertheless exceptions are raise but only the corresponding bits in the status work are set. This allows to use `fetestexcept` afterwards to test for the individual exceptions.

The new functions still are not the perfect solution. What is needed is that the generated signals can be intercepted and in the signal handler the user can specify how to proceed. The `siginfo_t` of the real-time signal extensions already provides a way to pass information about the offending instruction. But first, the Linux kernel does not (yet) support this. And second, it will be very hard to write code which allows to handle this situation.

To illustrate the problem we assume a division instruction which causes a division by zero exception. A signal handler is entered and one can learn about the address of the instruction. But what's next? There are several possibilities and one of them is to carry on with the program with using infinity (or minus infinity) as the result of the division. This is what an IEEE implementation has to do if a no signal is generated. But here the trouble starts. Each platform has its own way to implement this. The problems to be solved are:

- How to pass the result (infinity) back to the program? In most cases this is possible with a signal context but this is not guaranteed.
- When returning from the signal handler, where will the program resume? Some implementation might carry on with the next instruction, others might redo the instruction causing the exception. Again others might behave different based on the generated exception.

One can imagine how complicated the implementation is. This certainly means that library functions must be provided to implement the behaviour. Otherwise one would have to redo the complicated and non-portable work for every program. Just one example: on a platform where the instruction causing the exception is re-executed but one wants to simply set the result of the instruction and carry on, the signal handler will have to find out where the next instruction is. This requires understanding the machine code of the program and all the opcodes. This is hardly something everybody writing code using signal handlers will do her/himself.

Unfortunately the ISO C committee closed the discussion about numeric library extensions before coming to a conclusion about this problem and therefore the ISO C 9x library will not standardize any function of help for these situations. The `glibc` is intended to be extended with suitable functions in future but still it only helps on systems with `glibc`.

Slide 28

Extended Integer Types

ISO C 89 contains only a few integer types, all with machine dependent sizes.

$\Longrightarrow$ this makes it hard to write machine low-level and portable code.

ISO C 9x introduced new types to address this problem:

- exact-width types: `int8_t`, `int16_t`, `int32_t`, `int64_t`, `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`

- minimum-width types: `int_least8_t`, `int_least16_t`, `int_least32_t`, `int_least64_t`, `uint_least8_t`, `uint_least16_t`, `uint_least32_t`, `uint_least64_t`

- fastest-width types: `int_fast8_t`, `int_fast16_t`, `int_fast32_t`, `int_fast64_t`, `uint_fast8_t`, `uint_fast16_t`, `uint_fast32_t`, `uint_fast64_t`

Benefits of the new Types

For a long time in the history of programming with C the old types were sufficent for most purposes. Though the standard does not prescribe for the types `char`, `short`, `int`, and `long` any fixed size there was not much variation. All (interesting) platforms had 8bit bytes and were 32bits machines.

This changed with the introduction of 64bit machines. Suddenly people found themself in a self-afflicted dilemma: there programs won't compile and run without changes on the new machines. And even if they run they probably couldn't exchange data with the binaries on 32bit machines.

The October 1998 issues of *Performance Computing* contains an article by yours truly and Michael Tiemann about exactly this problem. It also describes the solutions and advice on how to write programs safe from these problems.

Slide 29

Extended Integer Types, II

- integer types to hold pointer values: `intptr_t`, `uintptr_t`

- greatest-width integer: `intmax_t`, `uintmax_t`

In addition all the needed limit macros are available (`INT8_MIN`, `UINT_LEAST64_MAX`, `INTPTR_MAX`, `UINTMAX_MAX`

Also available: macros to convert to minimum and maximum-width integers:

`INT8_C(`n`)` returns a representation of n suitable for a 8 bit constant. Important for types with size > `sizeof(int)`:

`UINT64_C(123456)` might expand to `123456ULL`

How to use the Extended Types

Having the new types is nice and one certainly will soon figure out where to use them. But variables of these types have to be used with some functions. One has to read and write such values and the old functions are not available anymore.

For this purpose there are a few new functions, following existing ones (defined in `inttypes.h`):

```
intmax_t imaxabs(intmax_t n);
imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
intmax_t strtoimax(const char *restrict nptr, char **restrict endptr, int base);
uintmax_t strtoumax(const char *restrict nptr, char **restrict endptr, int base);
intmax_t wcstoimax(const wchar_t *restrict nptr, wchar_t **restrict endptr, int base);
uintmax_t wcstoumax(const wchar_t *restrict nptr, wchar_t **restrict endptr, int base);
```

Beside this the `inttypes.h` header defines macros which help using `printf` and `scanf` with the new types. Every type can be read and written with the help of one of the 156 macros. The use of macros was the reason of many discussions in the ISO C committee and never was liked by anybody. But it was the one solution which would not conflict with other uses of the functions.

Printing of a unsigned 64bit int in octal form would be written like this:

```
uint64_t n;
n = ...;  /* Somehow determine a value.  */
printf ("n = %" PRIo64 "\n", n);
```

Reading a fast 16bit value in decimal notation would be written like this:

```
int_fast16_t n;
scanf ("%" SCNdFAST16, &n);
```

Especially this last example should show how important the proper use of these macros is. The example features the "fast 16bit" type. On 32bit platforms this often actually is a 32bit type. If one now would lazily write

```
scanf ("%hd", &n);
```

this would be very wrong since (assuming the 'h' modifier selects a 16bit type) the variable n would not be initialized properly.

Now, how are these macros implemented. Very easy, one only has to remember the rules os ISO C about string constants following each other: they get concatenated by the compiler. So the two macros above can be defined as

```
#define PRIo64          "llo"
#define SCNdFAST16      "d"
```

Just make sure you understand why this works in the above case. Also think about why the macros do not include the percent sign ('%') introducing the format element. The complete list of available formats is:

PRId8	PRId16	PRId32	PRId64		
PRIdLEAST8	PRIdLEAST16	PRIdLEAST32	PRIdLEAST64		
PRIdFAST8	PRIdFAST16	PRIdFAST32	PRIdFAST64		
PRIi8	PRIi16	PRIi32	PRIi64		
PRIiLEAST8	PRIiLEAST16	PRIiLEAST32	PRIiLEAST64		
PRIiFAST8	PRIiFAST16	PRIiFAST32	PRIiFAST64		
PRIo8	PRIo16	PRIo32	PRIo64		
PRIoLEAST8	PRIoLEAST16	PRIoLEAST32	PRIoLEAST64		
PRIoFAST8	PRIoFAST16	PRIoFAST32	PRIoFAST64		
PRIu8	PRIu16	PRIu32	PRIu64		
PRIuLEAST8	PRIuLEAST16	PRIuLEAST32	PRIuLEAST64		
PRIuFAST8	PRIuFAST16	PRIuFAST32	PRIuFAST64		
PRIx8	PRIx16	PRIx32	PRIx64		
PRIxLEAST8	PRIxLEAST16	PRIxLEAST32	PRIxLEAST64		
PRIxFAST8	PRIxFAST16	PRIxFAST32	PRIxFAST64		
PRIX8	PRIX16	PRIX32	PRIX64		
PRIXLEAST8	PRIXLEAST16	PRIXLEAST32	PRIXLEAST64		
PRIXFAST8	PRIXFAST16	PRIXFAST32	PRIXFAST64		
PRIdMAX	PRIiMAX	PRIoMAX	PRIuMAX	PRIxMAX	PRIXMAX
PRIdPTR	PRIiPTR	PRIoPTR	PRIuPTR	PRIxPTR	PRIXPTR
SCNd8	SCNd16	SCNd32	SCNd64		
SCNdLEAST8	SCNdLEAST16	SCNdLEAST32	SCNdLEAST64		
SCNdFAST8	SCNdFAST16	SCNdFAST32	SCNdFAST64		
SCNi8	SCNi16	SCNi32	SCNi64		
SCNiLEAST8	SCNiLEAST16	SCNiLEAST32	SCNiLEAST64		
SCNiFAST8	SCNiFAST16	SCNiFAST32	SCNiFAST64		
SCNo8	SCNo16	SCNo32	SCNo64		
SCNoLEAST8	SCNoLEAST16	SCNoLEAST32	SCNoLEAST64		
SCNoFAST8	SCNoFAST16	SCNoFAST32	SCNoFAST64		
SCNu8	SCNu16	SCNu32	SCNu64		
SCNuLEAST8	SCNuLEAST16	SCNuLEAST32	SCNuLEAST64		
SCNuFAST8	SCNuFAST16	SCNuFAST32	SCNuFAST64		
SCNx8	SCNx16	SCNx32	SCNx64		
SCNxLEAST8	SCNxLEAST16	SCNxLEAST32	SCNxLEAST64		
SCNxFAST8	SCNxFAST16	SCNxFAST32	SCNxFAST64		
SCNX8	SCNX16	SCNX32	SCNX64		
SCNXLEAST8	SCNXLEAST16	SCNXLEAST32	SCNXLEAST64		
SCNXFAST8	SCNXFAST16	SCNXFAST32	SCNXFAST64		
SCNdMAX	SCNiMAX	SCNoMAX	SCNuMAX	SCNxMAX	SCNXMAX
SCNdPTR	SCNiPTR	SCNoPTR	SCNuPTR	SCNxPTR	SCNXPTR

Slide 30

printf/scanf Extensions

New types require new modifiers:

- "hh": `char` argument
- "ll": `long long` argument
- "j": `intmax_t`, `uintmax_t` argument
- "z": `size_t` argument
- "t": `ptrdiff_t` argument

Often missed format:

- "a", "A": hexadecimal floating-point

Hexadecimal Floating-Point Constants

One weak point of the ISO C 89 standard when it comes to numeric programming is that it was not possible to specify exact floating-point numbers reliably, either in source code or when reading at runtime. Today's architectures all have floating-point numbers which use a representation in exponential form with a base of 2. But the format to write numbers is decimal. Therefore almost every number cannot specified exactly, one has to rely on rounding.

The hexadecimal format ends this dilemma. In source code one now can write:

```
double e = 0x1.5bf0a8b145769p+1;
```

At runtime the `printf`, `scanf`, and `strtod` functions provide the same support. The stream functions use the new format specifier "a" and "A", where the case of the letters in the output is determined by the case of the format character. If this format is used the number must be in the hexadecimal format. I.e., it does not mean that the hexadecimal format is accepted in addition to the decimal format.

This is different for the `strtod` function (and therefore for `strtof` and `strtold`). The function simply reads the next floating-point number, regardless of the format. It accepts the decimal and the hexadecimal format.

Slide 31

New Time Functions

`strftime`: 13 new formats

- Formats to represent date and time in ISO 8601 form.
  Was not possible before since no equivalence to "%z" (offset to DST) was available.
- three different formats to print the week number (necessary for monetary/fiscal applications)
- various different format which were felt missing.

`getdate`: parse date according to rules to be specified at runtime and return borken down time

ISO 8601

The world knows many different forms to write date and time. The Gregorian claendar is only one form and even when using this calendar there are different possibilities to write the same date. This is ofte very confusing.

In an attempt to standardize the way dates and times are noted an international standard was developed: ISO 8601. It is available at

`http://www.iso.ch/markete/8601.pdf`

Almost every modern program supports this format and many (most?) programs used internationally entirely uses this format. This includes many GNU programs.

The standard allows writing the date in several different form: with months and days, weeks and days, or simply day of the year. The most common form is

`YYYY-MM-DDThh:mm:ss+00:00`

The different parts are year, month, day, hour,minute, second, and offset from GMT in this order. When writing new programs which need textual representations of time and date it is highly advised to use the ISO 8601 format.

Parsing Dates

`glibc 2.0` already contained the function `strptime` which can be used to parse strings containing time and date information and store the result in an `struct tm` object. The problem with this function is that exactly one form of date and time is recognized at one time. One can use the locale's preferred form but this does not fundamentally change the problem. Dates especially are written in lots of different formats and recognizing only one can cause trouble.

For this reason System V systems provide a function `getdate` which is also standardized in Unix98 and therefore widely available. This function can parse a date and time string in arbitrary format as long as among the formats the user specifies at *runtime* there is a matching format. The formats which are tried must be made available in a file which is named by the environment variable `DATEMSK`.

Though this function potentially provides more functionality than `strptime` it has its own problems. The main one is that the user always has to provide a file with these format templates. Another drawback is that the use of the environment variable makes the function potentially unusable for multi-threaded applications.

Slide 32

IPv6 Functions

IPv6 proposal of the Internet Engineering Task Force and the POSIX.1g working groups extended the Basic Socket Interface:

- new data types `struct sockaddr_in6`
- interface name resolving and mapping:

  `unsigned int if_nametoindex (const char *ifname)`

  `char *if_indextoname (unsigned int ifindex, char *ifname)`

  `struct if_nameindex *if_nameindex (void)`

  `void if_freenameindex (struct if_nameindex *ptr)`
- Socket Address Structure to Nodename and Service Name:

```
int getnameinfo (const struct sockaddr *sa, socklen_t salen,
                 char *host, size_t hostlen,
                 char *serv, size_t servlen, int flags)
```

IPv6 Basic Socket Interface

The basic reading about this topic is

`http://www.ietf.org/internet-drafts/draft-ietf-ipngwg-bsd-api-new-06.txt`

We are constantly trying to catch up with the latest draft but mostly are behind since the changes are still substantial. It hopefully will get better and we have a stable interface.

Basically the new interfaces allow using the new IPv6 addressing (of course). But there are also many general enhancements made which are not only useful for IPv6. There is a set of functions which allow getting information about all the different interfaces a system has. This is something people haven't thought about in the past but which is very useful these days of machines with lots of interfaces.

The `getnameinfo` function allows more comfortable lookup capabilities. It provides reverse lookup of the hostname and the service name, both for IPv4 and IPv6.

`glibc 2.1` is known to not comply with the very latest draft. But we hope to have the missing functions implemented for a later version.

Slide 33

argp: Program Argument Parsing

- every commandline application has to parse the parameters
- POSIX.1's `getopt` provides basic functionality
- recurring work: handle options all programs have to have (`--help`, `--version`) and write help message
- not possible: have sub-parser for, e.g., libraries

⟹ the `argp` function family provides solutions for all that.

Using argp

Traditionally programs (at least those in GNU packages) consist of quite some code which is the same for all programs. All programs use `getopt` and they recognize some options: `--help` and `--version`. The output of `--help` is also a matter of repetition since it has to document all options. But those options are already specified in the `getopt` calls.

The `argp` function family reduce the work the programmer has to do. A typical example of a program using `argp` is this:

```
#include <argp.h>

const char *argp_program_version = "test-program";
const char *argp_program_bug_address = "<noone@nowhere.org>";

static char doc[] = "Simple description.";

/* A description of the arguments we accept.  */
static char args_doc[] = "ARG1 ARG2";

/* The options we understand.  */
static struct argp_option options[] = {
  {"verbose",  'v', 0,      0,  "Produce verbose output" },
  { 0 }
};

int verbose;

/* Parse a single option.  */
static error_t
parse_opt (int key, char *arg, struct argp_state *state)
{
```

```
  switch (key)
    {
    case 'v':
      verbose = 1;
      break;
    default:
      return ARGP_ERR_UNKNOWN;
    }
  return 0;
}

/* Our argp parser.  */
static struct argp argp = { options, parse_opt, args_doc, doc };

int main (int argc, char **argv)
{
  int remaining;

  /* Parse our arguments; every option seen by @code{parse_opt} will
     be reflected in @code{arguments}.  */
  argp_parse (&argp, argc, argv, 0, &remaining, NULL);

  do_work (argv[remaining], argv[remaining + 1], verbose);
  exit (0);
}
```

If one compares this code with the one necessary when using `getopt` one can see that the main differences are that we have no loop to process the arguments anymore. This happens in the `argp_parse` function. We also don't have to care at all about `--help`, `--version` and `--usage`. All these options are supported by default. Beside, the `argp` functions also automatically generate code for the `--help` function. The programmer therefore has not to care anymore about keeping the recognized options and the help text in sync.

More documentation of the `argp` functions can be found in the `glibc` manual. Read it online with

```
info libc 'Process Startup' 'Program Arguments' 'Parsing Program Arguments' Argp
```

Slide 34

Various GNU Extensions

`backtrace`: generate backtrace list for current point of execution

`printf_size`: `printf` extension which can be registered to print numbers in size units

`strverscmp`: similar to `strcmp` but embedded numbers are compared according to numeric value

`mempcpy`: similar to `memcpy` but return pointer to byte following the last written one

Backtracing

Every once in a while (especially when writing some kind of error handler) it is desirable to see how one gets to the current point of the execution. This most easily can be seen by a backtrace: the call chain of all open functions. The problem one normally has is that such a function is very unportable.

`glibc 2.1` provides three functions to help generating this output. They are both designed so that they can be used even in signal handlers for situation where the process memory might be currupted.

`int backtrace (void **array, int size)`

This function takes an array of at least `size * sizeof(void*)` bytes and stores the stackframe addresses for all open functions (excluding the `backtrace` call) in it. At most `size` elements are stored. Once one has retrieved this information it is possible to convert the informatin in readable form:

`char **backtrace_symbols (void *const *array, int size)`

This function returns a pointer to an array with `size` strings where each of the string is the textual representation of the information in the corresponding entry in `array`. Similarly:

`void backtrace_symbols_fd (void *const *array, int size, int fd)`

Instead of storing the information in a string it is written out in formatted form on the file descriptor `fd`. The output can look like this:

```
called from [0x8048558]
     /work/build/glibc-19990221/libc.so.6(bsearch+0x5b9) [0x4002063d]
     /work/build/glibc-19990221/libc.so.6(bsearch+0x55b) [0x400205df]
     /work/build/glibc-19990221/libc.so.6(bsearch+0x55b) [0x400205df]
     /work/build/glibc-19990221/libc.so.6(bsearch+0x55b) [0x400205df]
     /work/build/glibc-19990221/libc.so.6(qsort+0x3f) [0x40020793]
     [0x804851d]
     /work/build/glibc-19990221/libc.so.6(__libc_start_main+0xe6) [0x40018986]
     [0x8048471]
```

With the help of some tools from the `binutils` package it would even be possible to resolve the remaining non-symbolic references.

Sorting Strings with numbers

Quite often it is necessary to sort in programs strings which contain numbers. One omnipresent example is the `ls` program. GNU `ls` was therefore extended with an option `-v` which sorts the numbers while handling embedded numbers as if numeric. Example:

```
drepper@myware lwe# ls cm*
cmbx10.600pk   cmmi10.600pk  cmr7.600pk    cmsy10.600pk  cmtt10.600pk
cmbx12.600pk   cmmi9.600pk   cmr8.600pk    cmsy6.600pk   cmtt12.600pk
cmbx12.720pk   cmr10.600pk   cmr9.600pk    cmsy8.600pk   cmtt12.720pk
cmcsc10.600pk  cmr12.600pk   cmsl10.600pk  cmsy9.600pk   cmtt12.864pk
cmcsc10.864pk  cmr17.600pk   cmsl9.600pk   cmti10.600pk  cmtt9.600pk
```

This looks familiar but still nevertheless makes it hard to find the files easily since humans when looking at the files automatically take the numbers into account (not the single digits). With `ls`'s new option it is possible to get this result:

```
drepper@myware lwe# ls -v cm*
cmbx10.600pk   cmmi9.600pk   cmr10.600pk   cmsy6.600pk   cmtt9.600pk
cmbx12.600pk   cmmi10.600pk  cmr12.600pk   cmsy8.600pk   cmtt10.600pk
cmbx12.720pk   cmr7.600pk    cmr17.600pk   cmsy9.600pk   cmtt12.600pk
cmcsc10.600pk  cmr8.600pk    cmsl9.600pk   cmsy10.600pk  cmtt12.720pk
cmcsc10.864pk  cmr9.600pk    cmsl10.600pk  cmti10.600pk  cmtt12.864pk
```

The `glibc` contains a functions which can do this. By simply calling `strverscmp` instead of `strcmp` the result is just like in the example above.