Building and Using the Linux Software Development Environment

Hongjiu Lu
hjl@lucon.org
Lu Consulting
Santa Clara, CA, USA

March, 1996

Abstract

In this paper, we discuss the Linux software development environment. We describe how the each component from compiler, assembler, linker to the C library is built. We introduce some features in the Linux software development environment and the ways to use them.

1 Introduction

To develop software under Linux, we need compiler, assembler, linker, and the C library. When I first used the Linux kernel around 1990. There was only an old gcc 1.3x Linux port provided by Linus. The Linux C library was far from complete. It lacked many standard features and couldn't provide all functions which are necessary to build a complete Linux system. At the time, I had been using Unix for a while and played with the GNU tools while porting X386 to AT&T SVR3.2/386. While looking for an inexpensive Unix for my 386sx, I found out Linux on a Minix news group. Although there were only one bootdisk and one rootdisk, they were enough to convince me that it was the OS I want to put on my machine.

2 Software Development Environment

The C language itself doesn't provide intrinsic functions. The ANSI C standard specifies a set of C functions/macros which should be provided in the C library. The software development environment under Unix consists of **the C library**, **the C compiler**, **assembler** and **linker**.

2.1 The C Library

The Unix C library provides the functions specified by ANSI C standard and other functions needed for Unix operations. POSIX defines a standard way for an application program to obtain basic services from the operating system. Most parts of the Unix C library doesn't involve the Unix kernel. Only a limited set of services are implemented in the Unix kernel. The access to the Unix kernel from the application programs is provided via the system call interface.

When I first started using Linux, the Linux C library was very weak. The stdio library didn't conform the ANSI C standard. Many crucial functions for Unix operations were missing. Sometimes, it was an adventure to compile software packages obtained on the Internet.

2.2 The C Compiler

One import part of a Unix system is the C compiler. Most of the software available on the Internet are in the source code. Without a C compiler is it is very hard to get the software binary which can run under Unix.

Under Unix, usually the output of the C compiler is the assembly code. A bare bone C compiler takes the preprocessed C source code and generates the suitable assembly code. Although it is not required by the ANSI C standard, on most of Unix systems, a C preprocessor is used to processes header files, C language tokens and directives. To generate executable binary, we also need an assembler which generates the relocatable object code from the assembly code and a link editor which can output the binary in the format understood by the Unix kernel from the the relocatable object code.

The most visible C compiler command to the Unix programmers is **cc**, which is the C compiler driver. When **cc** is invoked, depending the command line option and the input file, it can generate the assembly source code, the relocatable object code or the executable binary.

3 Building Linux Software Development Environment

To build a Linux software development environment from scratch can be very hard if there are no source codes for C compiler, assembler and linker. Thanks to the GNU project of the Free Software Foundation, there are free GNU C compiler and binary utilities, which, among other things, include assembler and linker, for many CPU architectures. We can use the sources for those tools to build our own Linux software development environment. The most recent GNU C compiler and binary utilities source code can be found at

`ftp://prep.ai.mit.edu/pub/gnu`

The GNU C compiler source code is a tar file with filename of **gcc-x.y.z.tar.gz** and the binary utilities is **binutils-i.j.k.tar.gz**. That can be unpacked with

```
# gzip -dc gcc-x.y.z.tar.gz | tar xf -
# gzip -dc binutils-i.j.k.tar.gz | tar xf -
```

We should first choose a host machine to do our cross development on. The host platform should be supported by the GNU binary utilities and the GNU C compiler or at least it should be easy to add the support to binary utilities and gcc. Two most common host platforms are Sparc running SunOS 4.1.x and Intel x86 running Linux. We should be able to build any supported cross development tools under them without much trouble.

The next thing we have to decide is the binary format and the assembly code calling sequence. We have to choose a binary format supported by the Linux kernel. When I started building my own Linux software development environment for Intel x86 CPUs, only the **a.out** format was supported by the kernel. The a.out format has been used for many years under the BSD-derived operating systems. It is well understood and relatively easy to manipulate. Since then, we have moved to the **ELF** [1] format. It is much more flexible and powerful than the old a.out format.

3.1 The Assembler and Linker

I used an AT&T SVR3.2/386 to build a cross assembler and linker for Linux on Intel x86 in the a.out format with the GNU assembler 1.38 and the GNU binary utilities 1.9. At that time, there is no cross development support in assembler nor linker. I had to hard code many things to make the cross-assembler and cross-linker.

3.1.1 Configuration

The current GNU binutils fully support the cross development. We can configure it with

```
# configure --target=cpu-format
```

where **cpu** is our CPU architecture and **format** is our binary format. This command tells the binutils to configure itself to build a cross-assembler, a cross-linker and other cross binary utilities for our CPU in the specified format on the current host system. These cross binary utilities are independent of underlying operating system. As long as they run on the same CPU and use the same binary format, they can share the same set of binary utilities. If our CPU and binary format are supported by the GNU binary utilities, when the configure command returns.

3.1.2 Build and Installation

After configuration is done, we can start building the cross assembler and linker with

```
# make
```

The build should finish without any problem if our host machine and the target format is supported by the GNU binary utilities. When we install the cross assembler and linker, we have to consider the destination directory. We need to install the cross assembler and linker in a directory where they will be expected by the C compiler driver. We will leave that in Section 3.2 where we discuss building the GNU C compiler.

3.2 The GNU C Compiler

Before we start building the GNU C compiler, we need to determine what our C library has. That will determine how we configure the GNU C compiler. Besides the C library, some additional files are need for the C compiler to generate executable binaries.

3.2.1 Auxiliary Files

One common auxiliary file for the a.out binary format is the startup file, **crt0.o**. It is the first file the C compiler driver passes to the linker, **ld**. **crt0.o** sets up all the necessary stack variables and other bookkeeping procedures before transferring the execution to **main ()**.

When we use the profiling facility provided by the C compiler, we will use an alternative start-up file, **gcrt0.o**. In addition to what **crt0.o** does, **gcrt0.o** also setup calls to profiling routines such that when the executable runs a file will be generated with the profiling result.

The ELF binary format uses different auxiliary files, which are **crt1.o**, **crti.o**, **crtn.o**, **crtbegin.o** and **crtend.o**. crt1.o plays the similar role as crt0.o in the a.out binary format. **crti.o** and **crtn.o** are used to support the ELF's **.init** and **.fini** sections. **crtbegin.o** and **crtend.o** are used by the C++ compiler to support the file-scope constructors and destructors. For more information on **crti.o**, **crtn.o**, **crtbegin.o** and**crtend.o**, please read my another paper on ELF [1].

3.2.2 Configuration

The current GNU C compiler, 2.7.2, has excellent support for cross compilation. Fore a target platform on the CPU architecture running the operating system, we need to modify the configure script in the gcc source code to support

```
# configure --target=ourcpu-ouros
```

We need to provide at least two files used by the configure script, **tm_file**, the header file which describes our CPU/operating system and provide the gcc driver command line options, **tmake_file** which is the Makefile fragment needed for our target platform. **tmake_file** should contain

```
LIBGCC1=libgcc1.null
```

unless you can find a way to compile **libgcc1.a** which should be by the old C compiler. In our case, we are starting from the scratch. It won't be easy to compile **libgcc1.a** without an old C compiler.

3.2.3 Build and Installation

Before we start building the GNU C compiler for our target platform, we should build the GNU C compile for our host machine through the bootstrap normal procedure. At the time when start the build, we can just do

```
# make CC=gcc
```

The build process may not finish normally since we may not have our C library ready yet which will be compiled with the cross compiler we are building now. When the build stops, there should be a **cpp** which is a C preprocessor, **cc1** which is a C compiler, **cc1plus** which is a C++ compiler and **cc1obj** which is an Objective C compiler.

We have to install them by hand to the destination directory without **libgcc.a**, **libobjc.a**, **SYSCALLS.c.X**, nor **include/float.h**. **libgcc.a** and **libobjc.a** are only needed to build executables, which we cannot do until we have compiled our C library.

The cross compiler will look for assembler and linker in **$(prefix)/ourcpu–ouros/bin**. We have to make sure that we installed assembler, **as**, and linker, **ld**, in **$(prefix)/ourcpu–ouros/bin**. Otherwise, the cross compiler driver may be able to find the correct **as** and **ld**.

3.3 The Linux C Library

After the cross assemler, linker and C compiler are built, we are ready to compile our C library. At this pointer, we can use our newly built software development environment to compile our source code in C with one exception. We may have to write our system call interface in assembly code since jumping from the user space into the kernel space is out of the scope of the C compiler.

With some hard work, we cross-build our C library together with all the necessary auxillary files. Then we install the binaries files in **$(prefix)/ourcpu–ouros/lib** and the header files in **$(prefix)/ourcpu–ouros/include**. When it is done, we should go back to the GNU C compiler and this time

```
# make CC=gcc
```

should finish successfully if we set everything right. After installing **libgcc.a** and **libobjc.a**, a complete cross software development environment is built on our host machine. We can use it to cross-compile the software packages we want to run on our target machine including most of our kernel.

Since **include/float.h** is only CPU-dependent and almost OS-independent, we can use any **include/float.h** built under any OS under the same CPU. Or we can use the cross-compiled **enquire** to generate **include/float.h** on our target machine with

```
# ./enquire -f > include/float.h
```

We can even use the same gcc driver to compile for different targets. The **-b** option for **gcc** tells gcc for which target machine the output should be:

```
# gcc -b ourcpu-ouros .....
```

3.4 The Final Stage

With the previously built cross compile environment, we can compile binaries running on our Linux machine from our host platform. That is how the first gcc 2.x was built. Since I only had a slow Intel 386SX at the time, I crossed-compiled all the binaries I uploaded on the Linux gcc ftp sites, which were

```
ftp://tsx-11.mit.edu/pub/linux/packages/GCC
ftp://sunsite.unc.edu/pub/Linux/GCC
```

on none-Linux machines. It is relatively easy to cross compile most of software packages. We just need to make sure we use the right compiler, and the right **ar** programs plus any other binary utilties. **ar**, **ranlib** and other programs should be put in **$(prefix)/ourcpu–ouros/bin** along side with **as** and **ld**. We need to configure binary utiluties and gcc with

```
# configure --host=ourcpu-ouros --target=ourcpu-ouros --build=hostcpu-hostos
```

After the configuration is done, you should be able to cross compile the binary utilties and gcc with no or a few changes to Makefile.

With the native **as**, **ar**, **ld** and **ranlib**, it is very easy to recompile gcc and binary utilities under a Linux machine. On an Intell x86 machine, we just do

```
# configure  --prefix=/usr --local-prefix=/usr/local --gxx-include-dir=/usr/include/g++
```

to confiure gcc and then issue

```
# make bootstrap
# make compare
# make install
```

to build/install gcc from the source code.

For binary utilities, we just do

```
# configure  --prefix=/usr --enable-shared
```

to configure it. `--enable-shared` is used to build shared libraries for bfd and opcodes. To build and install, just do

```
# make
# make install
```

Through those straight foward steps, we can update our Linux software development tool set from thee source code once we cross-compiled our first C compiler, assembler and linker. One interesting note is when we moved from the a.out format to the ELF format, we simply cross-compiled from a.out to ELF until we got a working software development environment in ELF.

4 Using Linux Sofeware Development Environment

The Linux sofeware development environment consists of the GNU C compiler, the GNU binary utilities and the Linux C library. They are very similar to the typical Unix C sofeware development environment before the commercial Unix vendors unbundled the C compiler from their Unix operating systems. On Linux, all the source codes of those tools are available to the developers.

4.1 The Linux C Library

The Linux C library provides the basic functions to user applications. It is used in almost every Linux program. The C library is included and linked in when **cc** is used to compile a C source code.

The Linux C header files are designed for C and C++. No special cares are needed when we use the C header files for C++ programs. Also the same Linux C header file can be included more than once. Only the first occurrance will take effect.

4.1.1 BSD and System V

The design goal of the Linux C library is to be POSIX.1 compliant with many BSD and System V extensions. Most of BSD TCP/IP socket functions are supported under Linux. For certain functions, if POSIX.1 specifies both BSD and System V behaviors are acceptable, we favor the SVR4 if there is an equivelant BSD function in POSIX. If there are no conflicts, both BSD and System V header files and associated library functions are provided to help port softwares to Linux.

- The Linux **signal ()** and **setjmp ()/lonjmp ()** work like the System V ones. For the BSD behavior, the programer should use the POSIX functions **sigaction ()** and **sigsetjmp ()/siglongjmp ()**.

- The directory access is quite different before POSIX. The Linux C library provides header files for both POSIX and BSD implementations. But like POSIX, only the **d_name** field in the **dirent** structure should be used. The BSD **seekdir ()** and **telldir ()** functions are also included in the Linux C library. For the **DIR** pointer by **opendir ()**, it should be only used by subsequent **readdir ()**, **closedir ()**, **seekdir ()** and **telldir ()**. An access function **dirfd ()** is provided to access the file descriptor associated with the **DIR** pointer.
- The Linux C library uses the POSIX termios for terminal I/O, which is very close to the System V termio. For job control, the POSIX semantic should be used with terminal I/O and associated signals.

Like most of BSD systems, the Linux C library has a special function, **syscall ()**:

```
#include <syscall.h>

int syscall (int number, ...);
```

where **number** is the system call number. All the system calls number known to the current Linux C library are defined in **<sys/syscall.h>**. This fucntion can be used to access any system call interface. It is most often used by the new system call which is not implemented in the Linux C library yet. For example, we can use

```
#include <sys/types.h>
#include <syscall.h>

  int fd, whence, ret;
  loff_t offset;
  loff_t result;

  ...
  ret = syscall (140, fd, (off_t) (offset >> 32),
(off_t) (offset & 0xffffffff), &result, whence);
  ...
```

to access the system number 140 which is the long version of the **lseek ()** system call.

One thing should be noted here. The Linux C library header files depend on the kernel source code. The reason is all the system calls share the same data between kernel and user applications. It is natural for them to share the data. When we compile the user applications, we shoule be able to pick up whatever is supported in the kernel without update our C library header files.

4.2 Programming under Linux

The Linux C software development environment consists of the GNU tools and the Linux C library which is POSIX based with System V and BSD extension. This combination provides great flexibilites for proramming under Linux as well as porting to Linux.

4.2.1 Porting to Linux

For a software package, porting to Linux is just to pick up a supported target which is close to Linux. In general, any straight System V Release 4 (SVR4) based system should be a good start point to work on.

The Linux C library has the full support of BSD TCP/IP socket functions plus an enhanced 4.9.3 version of the BSD resolver library with one exception, although they are present in the Linux C library, **send ()**, **sendto ()**, **sendmsg ()**, **recv ()** , **recvfrom** and **recvmsg ()** haven't been completed in the Linux kernel yet. But this status can change at any time since people are working on improving Linux as we speak.

For the directory access, signal handling and terminal I/O [2], the Linux C library tries to follow POSIX.1. From my Linux porting experiences, I believe we have done a pretty good job on POSIX.

One issue we may face while compiling software under Linux is some softeware packages come with their own malloc libraries. On some Unix systems, the default system malloc library may not be very fast nor efficient. Some softeware developers provide an alternative malloc library to replace the default one. Under Linux, that is no longer true. Base on the best of my judgement, I have put the best malloc package available in the Linux C library. Theres should be no need for another malloc library under Linux. But under certain circumstances, we may want to use another malloc library. If it is the case, we have to be very careful about what are inside the alternative malloc library. The minimum requrement for a replacement malloc library is

```
extern void * malloc (size_t size);
extern void  free (void *ptr);
extern void  realloc (void *ptr, size size);
extern void  calloc (size_t nmemb, size size);
extern void * __libc_valloc (size_t size);
```

where **__libc_valloc ()** is similar to **malloc ()**. But it returns the allocated memory on the page boundary. It can be a performance concern. In certain cases, the memory has to be on the page boundary.

Another malloc related issue may come up during the porting. The Linux malloc library has very little tolerance on sloppy codes. If the software packages abuse the malloc library, they may core dump much faster under Linux than anyone Unix systems. I view that as a Linux feature. It should improve the qualities of softwares developed under Linux as well as those ported to Linux from other Unix systems.

One System V feature missing from Linux is **STREAM**. Another is Network Services Library, **libnsl**. They are used in SVR4 to implement TCP/IP. But the TCP/IP facility under Linux is implemented with a BSD socket interface. Fortunately most of free softwares are more or less BSD based. I have to yet to see a siginificant software depend on STREAM/**libnsl** to run. Usually it is not a problem.

4.3 Software Development under Linux

Since Linux heavily relies on the existing software packages on the Internet, one of the design goal of the Linux C library is to make porting as easy as possible in the meantime to follow the POSIX standard, the result is porting to Linux ususally is a matter of recompiling. When we write code under Linux, if we follow the POSIX standard, our software should be very easy to compile on any other modern Unix systems with the POSIX support.

4.3.1 C/C++ Compiler under Linux

Linux uses the GNU `C/C++` compilers as the default system compiler. **gcc** has many nice features make us life much easier under Linux.

One compiler flag is **-M**. It tells the preprocessor to generate a suitable for **make** describing the dependencies of each object file. We take a file, **foo.c**

```
# cat foo.c
#include <stdio.h>

int
main ()
{
  printf ("Hello World!\n");
  return 0;
}
```

We can use **gcc -M**

```
# gcc -M foo.c
foo.o: foo.c /usr/include/stdio.h /usr/include/features.h \
 /usr/include/sys/cdefs.h /usr/include/libio.h \
 /usr/include/_G_config.h
```

This feature also works on `C++` and any source files processed by the preprocessor, including the assembly source files with the **.S** suffix.

There are a few **-M** variants. Please consult the gcc manual page for details.

Another very useful flag is **-Wall**. It tells gcc to issue a warning if there are some potential problems in the source codes. You can also turn on the warning for spefici cases and/or add more **-W** flags to ask gcc to check more cases. It is always a good idea to compile the source code with **-Wall** and silence all the gcc warnings if possible.

4.3.2 Debugging and Profiling under Linux

To debug a user application under Linux, we need to compile the source code with the **-g** flag, e.g.:

```
# gcc -c -g foo.c
# gcc -g -o foo foo.o
```

We can use the GNU debugger **gdb** to debug the binary **foo** with

```
# gdb foo
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 960308 (i486-linux), Copyright 1996 Free Software Foundation, Inc...
(gdb) b main
Breakpoint 1 at 0x8048473: file foo.c, line 6.
(gdb) r
Starting program: /tmp/foo

Breakpoint 1, main () at foo.c:6
6   printf ("Hello World!\n");
(gdb) info shared
From        To          Syms Read   Shared Object Library
0x40008000  0x400b7f58  No          /lib/libc.so.5.3.6
0x40000000  0x400059bc  No          /lib/ld-linux.so.1
(gdb) c
Continuing.
Hello World!

Program exited normally.
(gdb) quit
#
```

To profile a user application, we need to compile it with

```
# gcc -c -p foo.c
# gcc -p -o foo foo.o
```

We use the GNU profile data display program **gprof** to produce an execution profile of **foo**.

```
# foo
Hello World!
# gprof foo
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00        1     0.00     0.00  main

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
   else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
   function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
   the function in the gprof listing. If the index is
   in parenthesis it shows where it would appear in
   the gprof listing if it were to be printed.

     Call graph (explanation follows)
```

```
granularity: each sample hit covers 4 byte(s) no time propagated

index % time    self  children    called     name
                0.00    0.00       1/1           _start [15]
[1]      0.0    0.00    0.00       1         main [1]
-----------------------------------------------

 This table describes the call tree of the program, and was sorted by
 the total amount of time spent in each function and its children.

 Each entry in this table consists of several lines.  The line with the
 index number at the left hand margin lists the current function.
 The lines above it list the functions that called this function,
 and the lines below it list the functions this one called.
 This line lists:
     index A unique number given to each element of the table.
Index numbers are sorted numerically.
The index number is printed next to every function name so
it is easier to look up where the function in the table.

     % time This is the percentage of the `total' time that was spent
in this function and its children.  Note that due to
different viewpoints, functions excluded by options, etc,
these numbers will NOT add up to 100%.

     self This is the total amount of time spent in this function.

     children This is the total amount of time propagated into this
function by its children.

     called This is the number of times the function was called.
If the function called itself recursively, the number
only includes non-recursive calls, and is followed by
a `+' and the number of recursive calls.

     name The name of the current function.  The index number is
printed after it.  If the function is a member of a
cycle, the cycle number is printed between the
function's name and the index number.


 For the function's parents, the fields have the following meanings:

     self This is the amount of time that was propagated directly
```

from the function into this parent.

children This is the amount of time that was propagated from the function's children into this parent.

called This is the number of times this parent called the function '/' the total number of times the function was called. Recursive calls to the function are not included in the number after the '/'.

name This is the name of the parent. The parent's index number is printed after it. If the parent is a member of a cycle, the cycle number is printed between the name and the index number.

If the parents of the function cannot be determined, the word '<spontaneous>' is printed in the 'name' field, and all the other fields are blank.

For the function's children, the fields have the following meanings:

self This is the amount of time that was propagated directly from the child into the function.

children This is the amount of time that was propagated from the child's children to the function.

called This is the number of times the function called this child '/' the total number of times the child was called. Recursive calls by the child are not listed in the number after the '/'.

name This is the name of the child. The child's index number is printed after it. If the child is a member of a cycle, the cycle number is printed between the name and the index number.

If there are any cycles (circles) in the call graph, there is an entry for the cycle-as-a-whole. This entry shows who called the cycle (as parents) and the members of the cycle (as children.) The '+' recursive calls entry shows the number of function calls that were internal to the cycle, and the calls entry for each member shows, for that member, how many times it was called from other members of the cycle.

```
Index by function name

   [1] main
#
```

The binary distribution of the Linux C library doesn't include **libg.a** nor **libc_p.a**. They are the debug version and profile version of the C library, respectively. Usually, those special C libraries are only needed by the C library developers. They are not required to debug nor profile the user applications.

To debug or profile the Linux C library, we need to compile the Linux C library ourselves and install **libg.a** and **libc_p.a**.

To debug the C library, we do

```
# gcc -c -ggdb foo.c
# gcc -ggdb -o foo foo.o
```

The **-ggdb** flag tells the compiler driver to link in the debug version of the C library so that we can debug it. With **gdb**, we can do

```
# gdb foo
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 960308 (i486-linux), Copyright 1996 Free Software Foundation, Inc...
(gdb) b write
Breakpoint 1 at 0x8052344
(gdb) r
Starting program: /tmp/foo

Breakpoint 1, 0x8052344 in __write ()
(gdb) bt
#0  0x8052344 in __write ()
#1  0x8050352 in _IO_file_write (f=0x8057670, data=0x805c000, n=13)
    at fileops.c:595
#2  0x804fe37 in _IO_do_write (fp=0x8057670, data=0x805c000 "Hello World!\n",
    to_do=13) at fileops.c:245
#3  0x8050015 in _IO_file_overflow (f=0x8057670, ch=10) at fileops.c:367
#4  0x80506f7 in __overflow (f=0x8057670, ch=10) at genops.c:162
#5  0x8050a85 in _IO_default_xsputn (f=0x8057670, data=0x8053198, n=13)
    at genops.c:351
#6  0x80504ac in _IO_file_xsputn (f=0x8057670, data=0x8053198, n=13)
```

```
    at fileops.c:687
#7  0x8048255 in _IO_vfprintf (s=0x8057670, format=0x8053198 "Hello World!\n",
    ap=0xbffffd48) at iovfprintf.c:180
#8  0x8048169 in _IO_printf (format=0x8053198 "Hello World!\n")
    at ioprintf.c:44
#9  0x804813d in main () at foo.c:6
#10 0x80480e4 in ___crt_dummy__ ()
(gdb) f 1
#1  0x8050352 in _IO_file_write (f=0x8057670, data=0x805c000, n=13)
    at fileops.c:595
595 {
(gdb) list
590 }
591
592 _IO_ssize_t
593 DEFUN(_IO_file_write, (f, data, n),
594       register _IO_FILE* f AND const void* data AND _IO_ssize_t n)
595 {
596   _IO_ssize_t to_do = n;
597   while (to_do > 0)
598     {
599       _IO_ssize_t count = _IO_write(f->_fileno, data, to_do);
(gdb) c
Continuing.
Hello World!

Program exited normally.
(gdb) quit
#
```

We can use the **-profile** to profile the Linux C library, e.g.:

```
# gcc -c -profile foo.c
# gcc -profile -o foo foo.o
```

For **foo** compiled with **gcc -profile**, we get

```
# foo
Hello World!
# gprof foo
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated
```

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00       14     0.00     0.00  _IO_file_overflow
  0.00      0.00     0.00       14     0.00     0.00  __overflow
  0.00      0.00     0.00        3     0.00     0.00  _IO_do_write
  0.00      0.00     0.00        3     0.00     0.00  __init_brk
  0.00      0.00     0.00        2     0.00     0.00  __libc_free
  0.00      0.00     0.00        2     0.00     0.00  __sbrk
  0.00      0.00     0.00        1     0.00     0.00  _IO_default_xsputn
  0.00      0.00     0.00        1     0.00     0.00  _IO_doallocbuf
  0.00      0.00     0.00        1     0.00     0.00  _IO_file_doallocate
  0.00      0.00     0.00        1     0.00     0.00  _IO_file_stat
  0.00      0.00     0.00        1     0.00     0.00  _IO_file_write
  0.00      0.00     0.00        1     0.00     0.00  _IO_file_xsputn
  0.00      0.00     0.00        1     0.00     0.00  _IO_printf
  0.00      0.00     0.00        1     0.00     0.00  _IO_setb
  0.00      0.00     0.00        1     0.00     0.00  _IO_vfprintf
  0.00      0.00     0.00        1     0.00     0.00  __default_morecore
  0.00      0.00     0.00        1     0.00     0.00  __default_morecore_init
  0.00      0.00     0.00        1     0.00     0.00  __init_dummy
  0.00      0.00     0.00        1     0.00     0.00  __isatty
  0.00      0.00     0.00        1     0.00     0.00  __libc_init
  0.00      0.00     0.00        1     0.00     0.00  __libc_malloc
  0.00      0.00     0.00        1     0.00     0.00  __libc_memalign
  0.00      0.00     0.00        1     0.00     0.00  __libc_valloc
  0.00      0.00     0.00        1     0.00     0.00  __new_exitfn
  0.00      0.00     0.00        1     0.00     0.00  __setfpucw
  0.00      0.00     0.00        1     0.00     0.00  __sigaction
  0.00      0.00     0.00        1     0.00     0.00  __tcgetattr
  0.00      0.00     0.00        1     0.00     0.00  _fxstat
  0.00      0.00     0.00        1     0.00     0.00  atexit
  0.00      0.00     0.00        1     0.00     0.00  exit
  0.00      0.00     0.00        1     0.00     0.00  main
  0.00      0.00     0.00        1     0.00     0.00  malloc_extend_top
  0.00      0.00     0.00        1     0.00     0.00  mbtowc

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
```

```
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
   else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
   function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
   the function in the gprof listing. If the index is
   in parenthesis it shows where it would appear in
   the gprof listing if it were to be printed.

     Call graph (explanation follows)


granularity: each sample hit covers 4 byte(s) no time propagated

index % time    self  children    called     name
                0.00    0.00       1/1           _start [214]
[1]      0.0    0.00    0.00       1         atexit [1]
                0.00    0.00       1/1           __new_exitfn [88]
-----------------------------------------------
                0.00    0.00       1/1           _start [214]
[2]      0.0    0.00    0.00       1         exit [2]
-----------------------------------------------
                0.00    0.00       1/1           _start [214]
[3]      0.0    0.00    0.00       1         main [3]
                0.00    0.00       1/1           _IO_printf [77]
-----------------------------------------------
                0.00    0.00       1/1           __libc_malloc [85]
[4]      0.0    0.00    0.00       1         malloc_extend_top [4]
                0.00    0.00       1/1           __default_morecore_init [81]
                0.00    0.00       1/1           __default_morecore [80]
-----------------------------------------------
                0.00    0.00       1/1           _IO_vfprintf [79]
[5]      0.0    0.00    0.00       1         mbtowc [5]
```

```
-----------------------------------------------
                0.00    0.00      14/14          __overflow [66]
[65]     0.0    0.00    0.00      14         _IO_file_overflow [65]
                0.00    0.00       2/3           _IO_do_write [67]
                0.00    0.00       1/1           _IO_doallocbuf [72]
-----------------------------------------------
                0.00    0.00       1/14          _IO_file_xsputn [76]
                0.00    0.00      13/14          _IO_default_xsputn [71]
[66]     0.0    0.00    0.00      14         __overflow [66]
                0.00    0.00      14/14          _IO_file_overflow [65]
-----------------------------------------------
                0.00    0.00       1/3           _IO_file_xsputn [76]
                0.00    0.00       2/3           _IO_file_overflow [65]
[67]     0.0    0.00    0.00       3         _IO_do_write [67]
                0.00    0.00       1/1           _IO_file_write [75]
-----------------------------------------------
                0.00    0.00       1/3           __default_morecore_init [81]
                0.00    0.00       2/3           __sbrk [70]
[68]     0.0    0.00    0.00       3         __init_brk [68]
-----------------------------------------------
                0.00    0.00       2/2           __libc_memalign [86]
[69]     0.0    0.00    0.00       2         __libc_free [69]
-----------------------------------------------
                0.00    0.00       1/2           __default_morecore [80]
                0.00    0.00       1/2           __default_morecore_init [81]
[70]     0.0    0.00    0.00       2         __sbrk [70]
                0.00    0.00       2/3           __init_brk [68]
-----------------------------------------------
                0.00    0.00       1/1           _IO_file_xsputn [76]
[71]     0.0    0.00    0.00       1         _IO_default_xsputn [71]
                0.00    0.00      13/14          __overflow [66]
-----------------------------------------------
                0.00    0.00       1/1           _IO_file_overflow [65]
[72]     0.0    0.00    0.00       1         _IO_doallocbuf [72]
                0.00    0.00       1/1           _IO_file_doallocate [73]
-----------------------------------------------
                0.00    0.00       1/1           _IO_doallocbuf [72]
[73]     0.0    0.00    0.00       1         _IO_file_doallocate [73]
                0.00    0.00       1/1           _IO_file_stat [74]
                0.00    0.00       1/1           __libc_valloc [87]
                0.00    0.00       1/1           _IO_setb [78]
                0.00    0.00       1/1           __isatty [83]
-----------------------------------------------
                0.00    0.00       1/1           _IO_file_doallocate [73]
```

```
[74]     0.0    0.00    0.00       1         _IO_file_stat [74]
                0.00    0.00       1/1           _fxstat [92]
-----------------------------------------------
                0.00    0.00       1/1           _IO_do_write [67]
[75]     0.0    0.00    0.00       1         _IO_file_write [75]
-----------------------------------------------
                0.00    0.00       1/1           _IO_vfprintf [79]
[76]     0.0    0.00    0.00       1         _IO_file_xsputn [76]
                0.00    0.00       1/14          __overflow [66]
                0.00    0.00       1/3           _IO_do_write [67]
                0.00    0.00       1/1           _IO_default_xsputn [71]
-----------------------------------------------
                0.00    0.00       1/1           main [3]
[77]     0.0    0.00    0.00       1         _IO_printf [77]
                0.00    0.00       1/1           _IO_vfprintf [79]
-----------------------------------------------
                0.00    0.00       1/1           _IO_file_doallocate [73]
[78]     0.0    0.00    0.00       1         _IO_setb [78]
-----------------------------------------------
                0.00    0.00       1/1           _IO_printf [77]
[79]     0.0    0.00    0.00       1         _IO_vfprintf [79]
                0.00    0.00       1/1           mbtowc [5]
                0.00    0.00       1/1           _IO_file_xsputn [76]
-----------------------------------------------
                0.00    0.00       1/1           malloc_extend_top [4]
[80]     0.0    0.00    0.00       1         __default_morecore [80]
                0.00    0.00       1/2           __sbrk [70]
-----------------------------------------------
                0.00    0.00       1/1           malloc_extend_top [4]
[81]     0.0    0.00    0.00       1         __default_morecore_init [81]
                0.00    0.00       1/3           __init_brk [68]
                0.00    0.00       1/2           __sbrk [70]
-----------------------------------------------
                0.00    0.00       1/1           __libc_init [84]
[82]     0.0    0.00    0.00       1         __init_dummy [82]
-----------------------------------------------
                0.00    0.00       1/1           _IO_file_doallocate [73]
[83]     0.0    0.00    0.00       1         __isatty [83]
                0.00    0.00       1/1           __tcgetattr [91]
-----------------------------------------------
                0.00    0.00       1/1           _start [214]
[84]     0.0    0.00    0.00       1         __libc_init [84]
                0.00    0.00       1/1           __init_dummy [82]
-----------------------------------------------
```

```
                0.00    0.00       1/1           __libc_memalign [86]
[85]     0.0    0.00    0.00       1         __libc_malloc [85]
                0.00    0.00       1/1           malloc_extend_top [4]
-----------------------------------------------
                0.00    0.00       1/1           __libc_valloc [87]
[86]     0.0    0.00    0.00       1         __libc_memalign [86]
                0.00    0.00       2/2           __libc_free [69]
                0.00    0.00       1/1           __libc_malloc [85]
-----------------------------------------------
                0.00    0.00       1/1           _IO_file_doallocate [73]
[87]     0.0    0.00    0.00       1         __libc_valloc [87]
                0.00    0.00       1/1           __libc_memalign [86]
-----------------------------------------------
                0.00    0.00       1/1           atexit [1]
[88]     0.0    0.00    0.00       1         __new_exitfn [88]
-----------------------------------------------
                0.00    0.00       1/1           _start [214]
[89]     0.0    0.00    0.00       1         __setfpucw [89]
-----------------------------------------------
                0.00    0.00       1/1           profil [56]
[90]     0.0    0.00    0.00       1         __sigaction [90]
-----------------------------------------------
                0.00    0.00       1/1           __isatty [83]
[91]     0.0    0.00    0.00       1         __tcgetattr [91]
-----------------------------------------------
                0.00    0.00       1/1           _IO_file_stat [74]
[92]     0.0    0.00    0.00       1         _fxstat [92]
-----------------------------------------------

 This table describes the call tree of the program, and was sorted by
 the total amount of time spent in each function and its children.

 Each entry in this table consists of several lines.  The line with the
 index number at the left hand margin lists the current function.
 The lines above it list the functions that called this function,
 and the lines below it list the functions this one called.
 This line lists:
     index A unique number given to each element of the table.
Index numbers are sorted numerically.
The index number is printed next to every function name so
it is easier to look up where the function in the table.

     % time This is the percentage of the `total' time that was spent
in this function and its children.  Note that due to
```

different viewpoints, functions excluded by options, etc, these numbers will NOT add up to 100%.

self This is the total amount of time spent in this function.

children This is the total amount of time propagated into this function by its children.

called This is the number of times the function was called. If the function called itself recursively, the number only includes non-recursive calls, and is followed by a '+' and the number of recursive calls.

name The name of the current function. The index number is printed after it. If the function is a member of a cycle, the cycle number is printed between the function's name and the index number.

For the function's parents, the fields have the following meanings:

self This is the amount of time that was propagated directly from the function into this parent.

children This is the amount of time that was propagated from the function's children into this parent.

called This is the number of times this parent called the function '/' the total number of times the function was called. Recursive calls to the function are not included in the number after the '/'.

name This is the name of the parent. The parent's index number is printed after it. If the parent is a member of a cycle, the cycle number is printed between the name and the index number.

If the parents of the function cannot be determined, the word '<spontaneous>' is printed in the 'name' field, and all the other fields are blank.

For the function's children, the fields have the following meanings:

self This is the amount of time that was propagated directly

```
from the child into the function.

     children This is the amount of time that was propagated from the
child's children to the function.

     called This is the number of times the function called
this child `/' the total number of times the child
was called.  Recursive calls by the child are not
listed in the number after the `/'.

     name This is the name of the child.  The child's index
number is printed after it.  If the child is a
member of a cycle, the cycle number is printed
between the name and the index number.

 If there are any cycles (circles) in the call graph, there is an
 entry for the cycle-as-a-whole.  This entry shows who called the
 cycle (as parents) and the members of the cycle (as children.)
 The `+' recursive calls entry shows the number of function calls that
 were internal to the cycle, and the calls entry for each member shows,
 for that member, how many times it was called from other members of
 the cycle.


Index by function name

  [71] _IO_default_xsputn     [80] __default_morecore     [66] __overflow
  [67] _IO_do_write           [81] __default_morecore_init [70] __sbrk
  [72] _IO_doallocbuf         [68] __init_brk             [89] __setfpucw
  [73] _IO_file_doallocate    [82] __init_dummy           [90] __sigaction
  [65] _IO_file_overflow      [83] __isatty               [91] __tcgetattr
  [74] _IO_file_stat          [69] __libc_free            [92] _fxstat
  [75] _IO_file_write         [84] __libc_init             [1] atexit
  [76] _IO_file_xsputn        [85] __libc_malloc           [2] exit
  [77] _IO_printf             [86] __libc_memalign         [3] main
  [78] _IO_setb               [87] __libc_valloc           [4] malloc_extend_top
  [79] _IO_vfprintf           [88] __new_exitfn            [5] mbtowc
#
```

More information about **gdb** and **gprof** can be found in the **gdb** [6] and **gprof** [7] manuals.

4.3.3 Linking under Linux

Linux uses the ELF [1] binary format. It is very flexible and provides many useful functionalities. One of them is shared library. With gcc, we can build shared C/C++ libraries with ease.

To build shared library, we need to compile the source code with Position-Independent-Code (PIC):

```
# gcc -fPIC -O -c libfoo.c
```

To build shared library, we do:

```
# gcc -shared -o libfoo.so libfoo.o
```

We can also build dependencies into a shared library by

```
# gcc -shared -o libfoo.so libfoo.o
```

We can also build dependencies into a shared library by

```
# gcc -shared -o libfoo.so libfoo.o -lbar
```

That causes **libfoo.so** depends on **libbar.so**. Whenever **libfoo.so** is used, **libbar.so** will also be searched. If libbar.so is not at the default locations, **/lib**, **/usr/lib** and **/usr/local/lib**, the linker may not be able to find **libbar.so** when it is seached implicitly by **-lfoo** evenif **-Ldir** is used. If it happens, we have a few choices:

- The **-rpath-link** flag. We can pass the library directory infomation through the linker **-rpath-link** flag:

  ```
  # gcc -o foo -lfoo -Wl,-rpath-link,dir1:dir2:....
  ```

  It will tell the linker to search **dir1**, **dir2**, ... for shared library dependencies.
- `LD_LIBRARY_PATH` is used by the dynamic linker to find shared libraries when the program is run. If it is set at link time, a native ELF linker will also use it to find shared libraries at link time.
- `LD_RUN_PATH` is used by the linker at link time. If the **-rpath** argument is not used, then the linker behaves as though **-rpath** `LD_RUN_PATH` were specified for the linker. In other words, it incorporates the environment variable into the program, where the dynamic linker will see it and use it when the program is run.

If you set `LD_RUN_PATH` at link time, then you do not need to set `LD_LIBRARY_PATH` at run time. The one we should use depends upon what we plan to do with the binaries–for example, whether we plan to distribute them to other machines which may have shared libraries in a different places. In general, `LD_RUN_PATH` should be the last resort. The preferred way is **-rpath-link** since `LD_LIBRARY_PATH` may have other side effects.

One useful option for linking is **-symbolic**. When it is used to build shared library,

```
# gcc -symbolic -shared -o libfoo.so libfoo.o
```

it will bind references to global symbols when building a shared shared library. Those global symbols available during building shared library will be resolved and won't be overriden at run time.

A side note, the newer gdb can also debug shared libraries if they are compiled with **-g**. It sometime is necessary to debug a shared library since the static one may not show the same bug.

For `C++` programs, we should normally use `c++` instead of `g++`. The difference between the two is shown here

```
# cat bar.cc
#include <iostream.h>

int
main ()
{
  cout << "Hello World!" << endl;
  return 0;
}
# c++ -v -o bar bar.cc
 gcc -v -o bar bar.cc -lstdc++ -lm
Reading specs from /usr/lib/gcc-lib/i486-linux/2.7.2.l.1/specs
gcc version 2.7.2.l.1
 /usr/lib/gcc-lib/i486-linux/2.7.2.l.1/cpp -lang-c++ -v -undef -D__GNUC__=2 -D__GNUG__=2 -D__cplusplus -D__GNUC_MINOR__=7 -D__ELF__ -Dunix -Di386 -Dlinux -D__ELF__ -D__unix__ -D__i386__ -D__linux__ -D__unix -D__i386 -D__linux -Asystem(unix) -Asystem(posix) -Acpu(i386) -Amachine(i386) -D__i486__ bar.cc /tmp/cca03976.ii
GNU CPP version 2.7.2.l.1 (i386 Linux/ELF)
#include "..." search starts here:
#include <...> search starts here:
 /usr/include/g++
 /usr/local/include
 /usr/i486-linux/include
 /usr/lib/gcc-lib/i486-linux/2.7.2.l.1/include
 /usr/include
End of search list.
 /usr/lib/gcc-lib/i486-linux/2.7.2.l.1/cc1plus /tmp/cca03976.ii -quiet -dumpbase bar.cc -version -o /tmp/cca03976.s
GNU C++ version 2.7.2.l.1 (i386 Linux/ELF) compiled by GNU C version 2.7.2.l.1.
 /usr/i486-linux/bin/as -V -Qy -o /tmp/cca039761.o /tmp/cca03976.s
GNU assembler version 960228 (i486-linux), using BFD version 2.6.0.10
 /usr/i486-linux/bin/ld -m elf_i386 -dynamic-linker /lib/ld-linux.so.1 -o bar /usr/lib/crt1.o /usr/lib/crti.o /usr/lib/gcc-lib/i486-linux/2.7.2.l.1/crtbegin.o -L/usr/lib/gcc-lib/i486-linux/2.7.2.l.1 -L/usr/i486-linux/lib /tmp/cca039761.o -lstdc++ -lm -lgcc -lc -lgcc /usr/lib/gcc-lib/i486-linux/2.7.2.l.1/crtend.o /usr/lib/crtn.o
# ldd bar
```

```
libstdc++.so.28 => /usr/lib/libstdc++.so.28.0.0
libm.so.5 => /lib/libm.so.5.0.5
libc.so.5 => /lib/libc.so.5.3.6
# bar
Hello World!
# g++ -v -o bar bar.cc
 gcc -v -o bar bar.cc -lg++ -lstdc++ -lm
Reading specs from /usr/lib/gcc-lib/i486-linux/2.7.2.1.1/specs
gcc version 2.7.2.1.1
 /usr/lib/gcc-lib/i486-linux/2.7.2.1.1/cpp -lang-c++ -v -undef -D__GNUC__=2 -D__GNUG__=2 -D__cplusplus -D__GNUC_MINOR__=7 -D__ELF__ -Dunix -Di386 -Dlinux -D__ELF__ -D__unix__ -D__i386__ -D__linux__ -D__unix -D__i386 -D__linux -Asystem(unix) -Asystem(posix) -Acpu(i386) -Amachine(i386) -D__i486__ bar.cc /tmp/cca03986.ii
GNU CPP version 2.7.2.1.1 (i386 Linux/ELF)
#include "..." search starts here:
#include <...> search starts here:
 /usr/include/g++
 /usr/local/include
 /usr/i486-linux/include
 /usr/lib/gcc-lib/i486-linux/2.7.2.1.1/include
 /usr/include
End of search list.
 /usr/lib/gcc-lib/i486-linux/2.7.2.1.1/cc1plus /tmp/cca03986.ii -quiet -dumpbase bar.cc -version -o /tmp/cca03986.s
GNU C++ version 2.7.2.1.1 (i386 Linux/ELF) compiled by GNU C version 2.7.2.1.1.
 /usr/i486-linux/bin/as -V -Qy -o /tmp/cca039861.o /tmp/cca03986.s
GNU assembler version 960228 (i486-linux), using BFD version 2.6.0.10
 /usr/i486-linux/bin/ld -m elf_i386 -dynamic-linker /lib/ld-linux.so.1 -o bar /usr/lib/crt1.o /usr/lib/crti.o /usr/lib/gcc-lib/i486-linux/2.7.2.1.1/crtbegin.o -L/usr/lib/gcc-lib/i486-linux/2.7.2.1.1 -L/usr/i486-linux/lib /tmp/cca039861.o -lg++ -lstdc++ -lm -lgcc -lc -lgcc /usr/lib/gcc-lib/i486-linux/2.7.2.1.1/crtend.o /usr/lib/crtn.o
# ldd bar
libg++.so.28 => /usr/lib/libg++.so.28.0.0
libstdc++.so.28 => /usr/lib/libstdc++.so.28.0.0
libm.so.5 => /lib/libm.so.5.0.5
libc.so.5 => /lib/libc.so.5.3.6
# bar
Hello World!
#
```

As we can see, **g++** includes **libg++.so** which is not needed unless classes in **libg++.so** are used in our `C++` programs.

More information about ELF, the GNU C compiler and the GNU linker can be found in my ELF paper [1], the gcc manual [4], and the GNU linker manual [5].

5 Conclusion

As we have shown above, thanks to the Free Software Foundation, Inc., Cygnus Support and countless GNU/Linux developers, Linux has grown from a hobby system to a full blown Unix system with many advanced features in `C/C++` software development environment which are previously only available under those modern commercial Unix systems. It still amazes me today that the work I have done at my free time has been shared by so many people. Linux has come to a point where people can make many wonderful things happen under it. We hope more and more GNU/Linux developers will take advantage of those powerful tools and bring GNU/Linux to another level.

References

[1] H.J. Lu, *ELF: From The Programmer's Perspective*, Linux & Internet '95, Berlin, Germany, 1995.

[2] Donald Lewine, *POSIX Programmer's Guide*, O'Reilly & Associate, Inc., 1991.

[3] *SunOS 5.3 Linker and Libraries Manual*, SunSoft, 1993.

[4] Richard M. Stallman, *Using and Porting GNU CC for Version 2.6*, Free Software Foundation, September 1994.

[5] Steve Chamberlain and Roland Pesch, *Using ld: The GNU linker, ld version 2*, Cygnus Support, January 1994.

[6] Richard M. Stallman and Cygnus Support, *Debugging with GDB: The GNU Source-Level Debugger*, Free Software Foundation, Inc., 1995

[7] Jay Fenlason and Richard Stallman, *The GNU Profiler*, Free Software Foundation, Inc., 1992