Linux 操作系统分析

2 GNU 开发工具链简介

陈香兰（xlanchen@ustc.edu.cn）

计算机应用教研室 @ 计算机学院
嵌入式系统实验室 @ 苏州研究院
中国科学技术大学

Spring 2011

“工欲善其事，必先利其器”

– – 《论语》

Outline

GNU tools

- GNU tools 和其他一些优秀的开源软件可以完全覆盖上述类型的软件开发工具。为了更好的开发软件系统，需要熟悉如下一些软件
  - GCC——GNU 编译器集
  - Binutils——辅助 GCC 的主要软件
  - Gdb——调试器
  - make——软件工程工具
  - diff, patch——补丁工具
  - CVS——版本控制系统
  - ...

Outline

GCC——The GNU Compiler Collection

- 不仅仅是 C 语言编译器
- 目前，GCC 可以支持多种高级语言，如
  - C、C++
  - ADA
  - Objective-C、Objective-C++
  - JAVA
  - Fortran
  - PASCAL

GCC 下的工具

- cpp —预处理器 GNU C 编译器在编译前自动使用 cpp 对用户程序进行预处理
- gcc —符合 ISO 等标准的 C 编译器
- g++ —基本符合 ISO 标准的 C++ 编译器
- gcj —GCC 的 java 前端
- gnat —GCC 的 GNU ADA 95 前端

GNU Tools—gcc

- gcc 是一个强大的工具集合，它包含了预处理器、编译器、汇编器、链接器等组件。它会在需要的时候调用其他组件。输入文件的类型和传递给 gcc 的参数决定了 gcc 调用具体的哪些组件。
- 对于开发者，它提供的足够多的参数，可以让开发者全面控制代码的生成，这对嵌入式系统级的软件开发非常重要

gcc 使用举例（1）

源程序

```
// gcctest.c

#include <stdio.h>

int main(void)
{
        int i,j;
        i=0;
        j=0;
        i=j+1;
        printf("Hello World!\n");
        printf("i=j+1=%d\n",i);
}
```

gcc 使用举例（2）

- 编译和运行

```
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ ls
gcctest.c
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ gcc -o gcctest gcctest.c
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ ls
gcctest  gcctest.c
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ ./gcctest
Hello World!
i=j+1=1
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$
```

gcc 的工作过程（1）

- 如果使用 -v 选项，则可以看到许多被隐藏的信息

```
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ gcc -o gcctest gcctest.c -v
使用内建 specs。
目标：i486-linux-gnu
配置为：../src/configure -v --with-pkgversion='Ubuntu 4.3.3-5ubuntu4' --with-bugurl=file:///usr/share/doc/gcc-4.3/README.
Bugs --enable-languages=c,c++,fortran,objc,obj-c++ --prefix=/usr --enable-shared --with-system-zlib --libexecdir=/usr/lib
 --without-included-gettext --enable-threads=posix --enable-nls --with-gxx-include-dir=/usr/include/c++/4.3 --program-suf
fix=-4.3 --enable-clocale=gnu --enable-libstdcxx-debug --enable-objc-gc --enable-mpfr --enable-targets=all --with-tune=ge
neric --enable-checking=release --build=i486-linux-gnu --host=i486-linux-gnu --target=i486-linux-gnu
线程模型：posix
gcc 版本 4.3.3 (Ubuntu 4.3.3-5ubuntu4)
COLLECT_GCC_OPTIONS='-o' 'gcctest' '-v' '-mtune=generic'
 /usr/lib/gcc/i486-linux-gnu/4.3.3/cc1 -quiet -v gcctest.c -D_FORTIFY_SOURCE=2 -quiet -dumpbase gcctest.c -mtune=generic
-auxbase gcctest -version -fstack-protector -o /tmp/cc3RRDBk.s
忽略不存在的目录“/usr/local/include/i486-linux-gnu”
忽略不存在的目录“/usr/lib/gcc/i486-linux-gnu/4.3.3/../../../../i486-linux-gnu/include”
忽略不存在的目录“/usr/include/i486-linux-gnu”
#include "..." 搜索从这里开始：
#include <...> 搜索从这里开始：
 /usr/local/include
 /usr/lib/gcc/i486-linux-gnu/4.3.3/include
 /usr/lib/gcc/i486-linux-gnu/4.3.3/include-fixed
 /usr/include
搜索列表结束。
GNU C (Ubuntu 4.3.3-5ubuntu4) 版本 4.3.3 (i486-linux-gnu)
        由 GNU C 版本 4.3.3 编译，  GMP 版本 4.2.4，MPFR 版本 2.4.0。
GGC 准则：--param ggc-min-expand=100 --param ggc-min-heapsize=131072
Compiler executable checksum: 0bf5703b57e064ca90b48f4e2c186f4a
COLLECT_GCC_OPTIONS='-o' 'gcctest' '-v' '-mtune=generic'
 as -V -Qy -o /tmp/ccKh3phy.o /tmp/cc3RRDBk.s
GNU assembler version 2.19.1 (i486-linux-gnu) using BFD version (GNU Binutils for Ubuntu) 2.19.1
COMPILER_PATH=/usr/lib/gcc/i486-linux-gnu/4.3.3/:/usr/lib/gcc/i486-linux-gnu/4.3.3/:/usr/lib/gcc/i486-linux-gnu/:/usr/lib
/gcc/i486-linux-gnu/4.3.3/:/usr/lib/gcc/i486-linux-gnu/:/usr/lib/gcc/i486-linux-gnu/4.3.3/:/usr/lib/gcc/i486-linux-gnu/
LIBRARY_PATH=/usr/lib/gcc/i486-linux-gnu/4.3.3/:/usr/lib/gcc/i486-linux-gnu/4.3.3/:/usr/lib/gcc/i486-linux-gnu/4.3.3/../.
./../../lib/:/lib/../lib/:/usr/lib/../lib/:/usr/lib/gcc/i486-linux-gnu/4.3.3/../../../:/lib/:/usr/lib/
COLLECT_GCC_OPTIONS='-o' 'gcctest' '-v' '-mtune=generic'
 /usr/lib/gcc/i486-linux-gnu/4.3.3/collect2 --eh-frame-hdr -m elf_i386 --hash-style=both -dynamic-linker /lib/ld-linux.so
.2 -o gcctest -z relro /usr/lib/gcc/i486-linux-gnu/4.3.3/../../../../lib/crt1.o /usr/lib/gcc/i486-linux-gnu/4.3.3/../../.
./../lib/crti.o /usr/lib/gcc/i486-linux-gnu/4.3.3/crtbegin.o -L/usr/lib/gcc/i486-linux-gnu/4.3.3 -L/usr/lib/gcc/i486-linu
x-gnu/4.3.3 -L/usr/lib/gcc/i486-linux-gnu/4.3.3/../../../../lib -L/lib/../lib -L/usr/lib/../lib -L/usr/lib/gcc/i486-linux
-gnu/4.3.3/../../.. /tmp/ccKh3phy.o -lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed
 /usr/lib/gcc/i486-linux-gnu/4.3.3/crtend.o /usr/lib/gcc/i486-linux-gnu/4.3.3/../../../../lib/crtn.o
```

gcc 的编译过程

- 一般情况下，c 程序的编译过程为
  1. 预处理
  2. 编译成汇编代码
  3. 汇编成目标代码
  4. 链接

1、预处理

- 预处理：使用 -E 参数
  输出文件的后缀为“.cpp”

  gcc -E -o gcctest.cpp gcctest.c

- 使用 wc 命令比较预处理后的文件与源文件，可以看到两个文件的差异

```
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ ls
gcctest.c
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ gcc -E -o gcctest.cpp gcctest.c
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ ls
gcctest.c  gcctest.cpp
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ wc gcctest.c gcctest.cpp
   18    17   149 gcctest.c
  761  1754 14508 gcctest.cpp
  779  1771 14657 总用量
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$
```

关于 wc 命令

```
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ wc --help
用法：wc [选项]... [文件]...
  或：wc [选项]... --files0-from=F
印出每个指定<文件>的行数、单词数和字节数的统计，如果指定了
多于一个<文件>，还会给出所有相关数据的总计。如果不指定
<文件>，或者<文件>为 - ，程序将从标准输入读取数据。
  -c, --bytes     输出字节数统计
  -m, --chars     输出字符数统计
  -l, --lines     输出行数统计
      --files0-from=F       从指定的文件F中读取输入文件，在F中
                    输入文件名以NUL结束
  -L, --max-line-length  显示最长的行
  -w, --words            显示单词数
      --help     显示此帮助信息并离开
      --version  显示版本信息并离开

请向 <bug-coreutils@gnu.org> 报告错误。
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$
```

2、编译成汇编代码

预处理文件 → 汇编代码

1. 使用**-x**参数说明根据指定的步骤进行工作，**cpp-output**指明从预处理得到的文件开始编译
2. 使用**-S**说明生成汇编代码后停止工作

gcc -x cpp-output -S -o gcctest.s gcctest.cpp

也可以直接编译到汇编代码

gcc -S gcctest.c

```
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ ls
gcctest.c  gcctest.cpp
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ gcc -x cpp-output -S -o gcctest.s gcctest.cpp
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ ls
gcctest.c  gcctest.cpp  gcctest.s
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ rm gcctest.cpp gcctest.s
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ ls
gcctest.c
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ gcc -S gcctest.c
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ ls
gcctest.c  gcctest.s
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$
```

3、编译成目标代码

汇编代码 $\longrightarrow$ 目标代码

gcc -x assembler -c gcctest.s

直接编译成目标代码

gcc -c gcctest.c

使用汇编器生成目标代码

as -o gcctest.o gcctest.s

```
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ ls
gcctest.c
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ gcc -S gcctest.c
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ ls
gcctest.c  gcctest.s
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ gcc -x assembler -c gcctest.s
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ ls
gcctest.c  gcctest.o  gcctest.s
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ rm gcctest.o gcctest.s
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ ls
gcctest.c
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ gcc -c gcctest.c
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ ls
gcctest.c  gcctest.o
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ rm gcctest.o
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ ls
gcctest.c
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ gcc -S gcctest.c
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ ls
gcctest.c  gcctest.s
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ as -o gcctest.o gcctest.s
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ ls
gcctest.c  gcctest.o  gcctest.s
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$
```

4、编译成执行代码

目标代码 $\longrightarrow$ 执行代码

gcc -o gcctest gcctest.o

直接生成执行代码

gcc -o gcctest gcctest.c

```
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ ls
gcctest.c  gcctest.o
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ gcc -o gcctest gcctest.o
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ ls
gcctest  gcctest.c  gcctest.o
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ ./gcctest
Hello World!
i=j+1=1
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ rm gcctest gcctest.o
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ gcc -o gcctest gcctest.c
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ ls
gcctest  gcctest.c
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ ./gcctest
Hello World!
i=j+1=1
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$
```

gcc 的高级选项

- -Wall：打开所有的警告信息

```
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ ls
gcctest.c
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ gcc -Wall -o gcctest gcctest.c
gcctest.c: 在函数‘main’中：
gcctest.c:13: 警告： 在有返回值的函数中，控制流程到达函数尾
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$
```

- 根据警告信息检查源程序

源程序

```
// gcctest.c

#include <stdio.h>

int main(void)
{
        int i,j;
        i=0;
        j=0;
        i=j+1;
        printf("Hello World!\n");
        printf("i=j+1=%d\n",i);
}
```

- **Main 函数的返回值为 int**
- **在函数的末尾应当添加返回一个值**

- 修改源程序

源程序 (修改)

```
// gcctest.c

#include <stdio.h>

int main(void)
{
    int i,j;
    i=0;
    j=0;
    i=j+1;
    printf("Hello World!\n");
    printf("i=j+1=%d\n",i);
    return 0;
}
```

```
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ ls
gcctest.c
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ gcc -Wall -o gcctest gcctest.c
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$ ls
gcctest  gcctest.c
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-exp$
```

优化编译

- 优化编译选项有：
  - -O0
    缺省情况，不优化
  - -O1
  - -O2
  - -O3
  - 等等

不同程度的优化
不同的优化内容

gcc 的优化编译举例（1）

考虑如下的源代码

//mytest.c

```
#include <stdio.h>
#include <math.h>
int main(void)
{
    int i,j;
    double k=0.0,k1=k2=k3=1.0;
    for (i=0;i<50000;i++)
        for (j=0;j<50000;j++)
        {
            k+=k1+k2+k3;
            k1 += 0.5;
            k2 += 0.2;
            k3 = k1+k2;
            k3 -= 0.1;
        }
    return 0;
}
```

- 使用不同的优化选项，分别生成不同的可执行文件

```
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-O$ ls
mytest.c
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-O$ gcc -O0 -o m0 mytest.c
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-O$ gcc -O1 -o m1 mytest.c
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-O$ gcc -O2 -o m2 mytest.c
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-O$ gcc -O3 -o m3 mytest.c
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-O$ ls
m0  m1  m2  m3  mytest.c
xlanchen@xlanchen-desktop:~/09FallEOS/GCC-O$
```

- 使用 time 命令统计程序的运行

```
time ./m0
time ./m1
time ./m2
time ./m3
```

- 一个以前机器上的结果

```
[donger@donger gcctest]$ ls
gcctest.c  m0  m1  m2  m3  mytest.c
[donger@donger gcctest]$ time ./m3

real    0m2.756s
user    0m2.658s
sys     0m0.042s
[donger@donger gcctest]$ time ./m2

real    0m2.733s
user    0m2.643s
sys     0m0.037s
[donger@donger gcctest]$ time ./m1

real    0m1.829s
user    0m1.767s
sys     0m0.022s
[donger@donger gcctest]$ time ./m0

real    0m40.808s
user    0m39.632s
sys     0m0.337s
[donger@donger gcctest]$
```

Outline

GNU binutils I

binutils 是一组二进制工具程序集，是辅助 GCC 的主要软件
主要包括

1. addr2line
   把程序地址转换为文件名和行号。在命令行中给它一个地址和一个可执行文件名，它就会使用这个可执行文件的调试信息指出在给出的地址上是哪个文件以及行号。
2. ar
   建立、修改、提取归档文件。归档文件是包含多个文件内容的一个大文件，其结构保证了可以恢复原始文件内容。
3. as
   是 GNU 汇编器，主要用来编译 GNU C 编译器 gcc 输出的汇编文件，他将汇编代码转换成二进制代码，并存放到一个 object 文件中，该目标文件将由连接器 ld 连接

GNU binutils II

4. C++filt
   解码 C++ 符号名，连接器使用它来过滤 C++ 和 Java 符号，防止重载函数冲突。
5. gprof
   显示程序调用段的各种数据。
6. ld
   是连接器，它把一些目标和归档文件结合在一起，重定位数据，并链接符号引用，最终形成一个可执行文件。通常，建立一个新编译程序的最后一步就是调用 ld。
7. nm
   列出目标文件中的符号。
8. objcopy
   把一种目标文件中的内容复制到另一种类型的目标文件中.

GNU binutils III

9. objdump
   显示一个或者更多目标文件的信息。使用选项来控制其显示的信息。它所显示的信息通常只有编写编译工具的人才感兴趣。
10. ranlib
    产生归档文件索引，并将其保存到这个归档文件中。在索引中列出了归档文件各成员所定义的可重分配目标文件。
11. readelf
    显示 elf 格式可执行文件的信息。
12. size
    列出目标文件每一段的大小以及总体的大小。默认情况下，对于每个目标文件或者一个归档文件中的每个模块只产生一行输出。

GNU binutils IV

13. strings
    打印某个文件的可打印字符串，这些字符串最少 4 个字符长，也可以使用选项 -n 设置字符串的最小长度。默认情况下，它只打印目标文件初始化和可加载段中的可打印字符；对于其它类型的文件它打印整个文件的可打印字符，这个程序对于了解非文本文件的内容很有帮助。
14. strip
    丢弃目标文件中的全部或者特定符号。
15. libiberty
    包含许多 GNU 程序都会用到的函数，这些程序有：getopt, obstack, strerror, strtol 和 strtoul.
16. libbfd
    二进制文件描述库.

GNU binutils V

17. libopcodes
    用来处理 opcodes 的库, 在生成一些应用程序的时候也会用到它, 比如 objdump.Opcodes 是文本格式可读的处理器操作指令.

binutils 开发工具使用举例

1. ar
2. nm
3. Objcopy
4. Objdump
5. readelf

1、ar

- ar 用于建立、修改、提取归档文件 (archive)，一个归档文件，是包含多个被包含文件的单个文件（也可以认为归档文件是一个库文件）。
- 被包含的原始文件的内容、权限、时间戳、所有者等属性都保存在归档文件中，并且在提取之后可以还原

使用 ar 建立库文件（1）

源程序 add.c

```
// add.c
int Add(int a, int b)
{
    int result;
    result = a+b;
    return result;
}
```

源程序 minus.c

```
// minus.c
int Minus(int a, int b)
{
    int result;
    result = a-b;
    return result;
}
```

- Step1：生成 **add.o** 和 **minus.o** 两个目标文件

```
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$ ls
add.c  minus.c  test.c
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$ gcc -c add.c minus.c
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$ ls
add.c  add.o  minus.c  minus.o  test.c
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$
```

- step2：生成库文件，并复制到 **/usr/lib/** 目录下

关于 ar 命令：

r - replace existing or insert new file(s) into the archive

v - be verbose

```
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$ ls
add.c  add.o  minus.c  minus.o  test.c
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$ ar rv libtest.a add.o minus.o
ar: creating libtest.a
a - add.o
a - minus.o
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$ ls
add.c  add.o  libtest.a  minus.c  minus.o  test.c
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$ sudo cp libtest.a /usr/lib/
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$
```

库文件使用举例: 在代码中使用 Add 和 Minus 函数

源代码 test.c

```
// test.c
#include <stdio.h>

int main(void)
{
        int a=8;
        int b=3;
        printf("a=%d\tb=%d\n",a,b);
        int sum = Add(a,b);
        printf("a+b=%d\n",sum);
        int cha = Minus(a,b);
        printf("a-b=%d\n",cha);
        return 0;
}
```

在编译时指定库文件

- 在链接时，使用“-l<name>”选项来指明库文件

```
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$ ls
add.c  minus.c  test.c
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$ gcc -o test test.c -ltest
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$ ls
add.c  minus.c  test  test.c
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$ ./test
a=8     b=3
a+b=11
a-b=5
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$
```

2、nm

- nm 的主要功能是列出目标文件中的符号，这样程序员就可以定位和分析执行程序和目标文件中的符号信息和它的属性

nm 显示的符号类型

- A：符号的值是绝对值，并且不会被将来的链接所改变
- B：符号位于未初始化数据部分（BSS 段）
- C：符号是公共的。公共符号是未初始化的数据。在链接时，多个公共符号可能以相同的名字出现。如果符号在其他地方被定义，则该文件中的这个符号会被当作引用来处理
- D：符号位于已初始化的数据部分
- T：符号位于代码部分
- U：符号未被定义 ？：符号类型未知，或者目标文件格式特殊

nm 使用举例

```
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$ ls
add.c  add.o  minus.c  minus.o  test  test.c
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$ gcc -c test.c
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$ ls
add.c  add.o  minus.c  minus.o  test  test.c  test.o
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$ nm test.o
         U Add
         U Minus
00000000 T main
         U printf
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$ nm add.o
00000000 T Add
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$ nm minus.o
00000000 T Minus
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$
```

作业：使用 nm 列出可执行文件 test 的符号，与 test.o 进行比较

3、objcopy

- 可以将一种格式的目标文件内容进行转换，并输出为另一种格式的目标文件。
- 它使用 GNU BFD(binary format description) 库读 / 写目标文件，通过这个 BFD 库，objcopy 能以一种不同于源目标文件的格式生成新的目标文件

objcopy -h

- 在 makefile 里面用**-O binary** 选项来生成原始的二进制文件，即通常说的 image 文件

Objcopy 使用举例

- 将 test 转换为 srec 格式的文件 ts
- 使用 file 命令查看文件信息

```
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$ ls
add.c  add.o  minus.c  minus.o  test  test.c  test.o
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$ file test
test: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically
 linked (uses shared libs), for GNU/Linux 2.6.15, not stripped
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$ objcopy -O srec test ts
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$ ls
add.c  add.o  minus.c  minus.o  test  test.c  test.o  ts
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$ file ts
ts: Motorola S-Record; binary data in text format
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$
```

文件格式 I

- a.out：
  assembler and link editor output
  汇编器和链接编辑器的输出
- coff
  common object file format
  一种通用的对象文件格式
- ELF
  excutive linked file
  Linux 系统所采用的一种通用文件格式，支持动态连接。
  ELF 格式可以比 COFF 格式包含更多的调试信息

文件格式 II

- Flat
  elf 格式有很大的文件头，flat 文件对文件头和一些段信息做了简化
  μClinux 系统使用 flat 可执行文件格式
- SREC
  MOTOROLA S-Recoder 格式（S 记录格式文件）
  等等

4、objdump

- 显示一个或多个目标文件的信息，由其选项来控制显示哪些信息。
- 一般来说，objdump 只对那些要编写编译工具的程序员有帮助，但是我们通过这个工具可以方便的查看执行文件或者库文件的信息

Objdump 使用举例（1）

1. “-f”选项显示文件头的内容

```
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$ ls
add.c  add.o  minus.c  minus.o  test  test.c  test.o  ts
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$ objdump -f test

test:     file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x08048310

xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$ objdump -f ts

ts:     file format srec
architecture: UNKNOWN!, flags 0x00000000:

start address 0x08048310

xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$
```

Objdump 使用举例（2）

- “-d”选项进行反汇编

```
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$ objdump -d add.o

add.o:     file format elf32-i386


Disassembly of section .text:

00000000 <Add>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 ec 10                sub    $0x10,%esp
   6:   8b 55 0c                mov    0xc(%ebp),%edx
   9:   8b 45 08                mov    0x8(%ebp),%eax
   c:   01 d0                   add    %edx,%eax
   e:   89 45 fc                mov    %eax,-0x4(%ebp)
  11:   8b 45 fc                mov    -0x4(%ebp),%eax
  14:   c9                      leave
  15:   c3                      ret
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$
```

5、readelf

- readelf：显示一个或多个 ELF 格式的目标文件信息。
- Readelf 使用举例

```
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$ readelf -h test
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048310
  Start of program headers:          52 (bytes into file)
  Start of section headers:          6068 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         8
  Size of section headers:           40 (bytes)
  Number of section headers:         36
  Section header string table index: 33
xlanchen@xlanchen-desktop:~/09FallEOS/ar_exp$
```

Outline

GNU Toolchain—gdb

- Gdb = GNU debuger
- GNU tools 中的调试器，功能强大
  - 设置断点
  - 监视、修改变量
  - 单步执行
  - 显示 / 修改寄存器的值
  - 堆栈查看
  - 远程调试

gdb 使用举例

源代码 bug.c

```
// bug.c

#include <stdio.h>
#include <stdlib.h>

static char buff[256];
static char* string;

int main(void)
{
    printf("input a string:");
    gets(string);
    printf("\n Your string is:%s\n",string);
    return 0;
}
```

编译并运行

```
xlanchen@xlanchen-desktop:~/09FallEOS/gdb_exp$ ls
bug.c
xlanchen@xlanchen-desktop:~/09FallEOS/gdb_exp$ gcc -o bug bug.c
/tmp/cc0s41nq.o: In function `main':
bug.c:(.text+0x26): warning: the `gets' function is dangerous and should no
t be used.
xlanchen@xlanchen-desktop:~/09FallEOS/gdb_exp$ ls
bug  bug.c
xlanchen@xlanchen-desktop:~/09FallEOS/gdb_exp$ ./bug
input a string:hello
段错误
xlanchen@xlanchen-desktop:~/09FallEOS/gdb_exp$
```

使用 gdb 调试 bug

```
xlanchen@xlanchen-desktop:~/09FallEOS/gdb_exp$ ls
bug  bug.c
xlanchen@xlanchen-desktop:~/09FallEOS/gdb_exp$ gdb bug
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
(gdb) run
Starting program: /home/xlanchen/09FallEOS/gdb_exp/bug
input a string:hello

Program received signal SIGSEGV, Segmentation fault.
0xb7e85c2a in gets () from /lib/tls/i686/cmov/libc.so.6
(gdb) where
#0  0xb7e85c2a in gets () from /lib/tls/i686/cmov/libc.so.6
#1  0x0804841e in main ()
(gdb)
```

能否查看源代码？

- 使用 gcc 的 -g 参数

gcc -g -o bug bug.c

- 重新调试

```
(gdb) run
Starting program: /home/xlanchen/09FallEOS/gdb_exp/bug
input a string:hello

Program received signal SIGSEGV, Segmentation fault.
0xb7ef1c2a in gets () from /lib/tls/i686/cmov/libc.so.6
(gdb) where
#0  0xb7ef1c2a in gets () from /lib/tls/i686/cmov/libc.so.6
#1  0x0804841e in main () at bug.c:12
(gdb) list
2
3       #include <stdio.h>
4       #include <stdlib.h>
5
6       static char buff[256];
7       static char* string;
8
9       int main(void)
10      {
11              printf("input a string:");
(gdb)
```

```
10      {
11              printf("input a string:");
(gdb) break 12
Breakpoint 1 at 0x8048411: file bug.c, line 12.
(gdb) next
Single stepping until exit from function gets,
which has no line number information.

Program terminated with signal SIGSEGV, Segmentation fault.
The program no longer exists.
(gdb) run
Starting program: /home/xlanchen/09FallEOS/gdb_exp/bug

Breakpoint 1, main () at bug.c:12
12              gets(string);
(gdb) next
input a string:hello

Program received signal SIGSEGV, Segmentation fault.
0xb7e56c2a in gets () from /lib/tls/i686/cmov/libc.so.6
(gdb) print string
$1 = 0x0
(gdb)
```

Outline

使用 GNU make 管理项目 GNU I

- make 是一种代码维护工具，在使用 GNU 编译器开发大型应用时，往往要使用 make管理项目。
  - 如果不使用 make 管理项目，就必须重复使用多个复杂的命令行维护项目和生成目标代码。
- Make 通过将命令行保存到 makefile 中简化了编译工作。
- Make 的主要任务是根据 makefile 中定义的规则和步骤，根据各个模块的更新情况，自动完成整个软件项目的维护和代码生成工作。
- Make 可以识别出 makefile 中哪些文件已经被修改，并且在再次编译的时候只编译这些文件，从而提高编译的效率

使用 GNU make 管理项目 GNU II

- Make 会检查文件的修改和生成时间戳，如果目标文件的修改或者生成时间戳比它的任意一个依赖文件旧，则 make 就执行 makefile 文件中描述的相应命令，以便更新目的文件
- 只更新那些需要更新的文件，而不重新处理那些并不过时的文件

- 特点：
  - 适合于支持多文件构成的大中型软件项目的编译，链接，清除中间文件等管理工作
  - 提供和识别多种默认规则，方便对大型软件项目的管理
  - 支持对多目录的软件项目进行递归管理
  - 对软件项目具有很好的可维护性和扩展性

makefile

- Makefile 告诉 make 该做什么、怎么做
- makefile 主要定义了
  1. 依赖关系
     即有关哪些文件的最新版本是依赖于哪些别的文件产生或者组成的
  2. 需要用什么命令来产生目标文件的最新版本
  3. 以及一些其他的功能

Makefile 的规则

规则

- 一条规则包含 3 个方面的内容，
  1. 要创建的目标（文件）
  2. 创建目标（文件）所依赖的文件列表；
  3. 通过依赖文件创建目标文件的命令组

规则一般形式

target ... : prerequisites ...
<tab>command
<tab>...
<tab>...

例如

test:test.c;gcc -O -o test test.c

一个简单的 makefile

```
edit : main.o kbd.o command.o display.o insert.o \
        search.o files.o utils.o
cc -o edit main.o kbd.o command.o display.o insert.o \
        search.o files.o utils.o

main.o : main.c defs.h
cc -c main.c

kbd.o : kbd.c defs.h command.h
cc -c kbd.c

command.o : command.c defs.h command.h
cc -c command.c

display.o : display.c defs.h buffer.h
cc -c display.c

insert.o : insert.c defs.h buffer.h
cc -c insert.c

search.o : search.c defs.h buffer.h
cc -c search.c

files.o : files.c defs.h buffer.h command.h
cc -c files.c

utils.o : utils.c defs.h
cc -c utils.c

clean :
rm edit main.o kbd.o command.o display.o insert.o \
        search.o files.o utils.o
```

Make 的工作过程

- default goal
  - 在缺省的情况下，make 从 makefile 中的第一个目标开始执行
- Make 的工作过程类似一次深度优先遍历过程

Makefile 中的变量

- 使用变量可以
  - 降低错误风险
  - 简化 makefile

例：objects 变量（$(objects)）

```
objects = main.o kbd.o command.o \
        display.o insert.o search.o files.o utils.o
edit : $(objects)
        cc -o edit $(objects)
```

- 有点像环境变量
  - 环境变量在 make 过程中被解释成 make 的变量
- 可以被用来
  - 贮存一个文件名列表。
  - 贮存可执行文件名。如用变量代替编译器名。
  - 贮存编译器 FLAG

预定义变量

- Make 使用了许多预定义的变量，如
  - AR
  - AS
  - CC
  - CXX
  - CFLAGS
  - CPPFLAGS
  - 等等

简化后的 makefile 文件

```
objects = main.o kbd.o command.o display.o \
        insert.o search.o files.o utils.o

edit : $(objects)
cc -o edit $(objects)

main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

.PHONY : clean
clean :
        rm edit $(objects)
```

内部变量

- $@ 扩展成当前规则的目的文件名
- $< 扩展成依赖列表中的第一个依赖文件
- $^扩展成整个依赖列表（除掉了里面所有重复的文件名）
- 等等

- 不需要括号括住

例如：

```
CC = gcc
CFLAGS = -Wall -O -g
foo.o : foo.c foo.h bar.h
    $(CC) $(CFLAGS) -c $< -o $@
```

隐含规则 (Implicit Rules)

- 内置的规则
- 告诉 make 当没有给出某些命令的时候，应该怎么办。
- 用户可以使用预定义的变量改变隐含规则的工作方式，如
  - 一个C 编译的具体命令将会是：

```
$(CC) $(CFLAGS) $(CPPFLAGS)
$(TARGET_ARCH) -c $< -o $@
```

设定目标（Phony Targets）

- 设定目标
  - 目标不是一个文件
  - 其目的是为了让一些命令得以执行
- 使用**PHONY**显式声明设定目标

```
.PHONY: clean
```

- 使用设定目标实现多个目的

```
all: prog1 prog2
```

典型的设定目标

- 设定目的也可以用来描述一些其他的动作。例如，想把中间文件和可执行文件删除，可以在 makefile 里设立这样一个规则：

```
clean:
        rm *.o exec_file
```

前提是没有其它的规则依靠这个 'clean' 目的，它将永远不会被执行。但是，如果你明确的使用命令 'make clean'，make 会把这个目的做为它的主要目标，执行那些 rm 命令

Makefile 中的函数 (Functions) I

- 用来计算出要操作的文件、目标或者要执行的命令
- 使用方法：

$$\$(function\ arguments)$$

- 典型的函数

$$\$(subst\ from,to,text)$$

$$\$(subst\ ee,EE,feet\ on\ the\ street)$$

  - 相当于 'fEEt on the strEEt'

Makefile 中的函数 (Functions) II

$(patsubst pattern,replacement,text)

$(patsubst %.c,%.o,x.c.c bar.c)

- 相当于 ‘x.c.o bar.o ‘

Makefile 中的函数 (Functions) III

$(wildcard pattern)

*$(wildcard *.c)*

- objects := $(wildcard *.o)

makefile 中的条件语句

conditional-directive
text-if-true
endif

- or

conditional-directive
text-if-true
else
text-if-false
endif

四种条件语句

- ifeq...else...endif
- ifneq···else···endif
- ifndef···else···endif
- ifndef...else···endif

实际项目中的 makefile s

找到 Linux 源代码中所有的 makefile，分析它们的功能、相互关系。

Outline

GNU Tools—ld

- ld，The GNU Linker
  Linux 上常用的链接器
- ld 软件的作用是把各种目标文件（.o 文件）和库文件链接在一起，并定位数据和函数地址，最终生成可执行程序
- gcc 可以间接的调用 ld，使用 gcc 的 -Wl 参数可以传递参数给 ld
- 使用下面的命令可以列出 ld 常用的一些选项：

ld –help

ld 使用举例 I

源程序

//hello.c

```
#include <stdio.h>
int main(void)
{
        printf("\n\nHello World!\n\n");
}
```

编译 hello.c 到 hello.o

gcc -c hello.c

链接

ld 使用举例 II

ld -dynamic-linker /lib/ld-linux.so.2 /usr/lib/crt1.o /usr/lib/crti.o /usr/lib/crtn.o hello.o -lc -o hello

运行

./hello

目标文件

- ld 通过 BFD 库可以读取和操作 coff、elf、a.out 等各种执行文件格式的目标文件
  - BFD（Binary File Descriptor）
- 目标文件（object file）
  - 由多个节 (section) 组成，常见的节有：
    - **text**节保存了可执行代码，
    - **data**节保存了有初值的全局标量，
    - **bss**节保存了无初值的全局变量。

使用 objdump 查看目标文件的信息

objdump -h hello.o

```
xlanchen@xlanchen-desktop:~/09FallEOS/ld_exp$ objdump -h hello.o

hello.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000026  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  00000000  00000000  0000005c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  0000005c  2**2
                  ALLOC
  3 .rodata       00000010  00000000  00000000  0000005c  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .comment      00000024  00000000  00000000  0000006c  2**0
                  CONTENTS, READONLY
  5 .note.GNU-stack 00000000  00000000  00000000  00000090  2**0
                  CONTENTS, READONLY
xlanchen@xlanchen-desktop:~/09FallEOS/ld_exp$
```

作业：比较 hello

链接描述文件（Linker script ）

- 可以使用链接描述文件控制 ld 的链接过程。

链接描述文件，command file

又称为链接脚本，Linker script

- 用来控制 ld 的链接过程
  - 描述各输入文件的各节如何映射到输出文件的各节
  - 控制输出文件中各个节或者符号的内存布局
- 使用的语言为：
  - The ld command language，链接命令语言
- ld 命令的**-T commandfile**选项指定了链接描述文件名
  - 如果不指定链接描述文件，ld 就会使用一个默认的描述文件来产生执行文件

作业：找到 Linux 的链接描述文件并分析。

链接描述文件的命令

- 链接描述文件的命令主要包括如下几类：
  - 设置入口点命令
  - 处理文件的命令
  - 处理文件格式的命令
  - 其他

常用的命令

- **设置入口点**

格式：

ENTRY(symbol)

  - 设置 symbol 的值为执行程序的入口点。

- ld 有多种方法设置执行程序的入口点，确定程序入口点的顺序如下：
  - ld 命令的**-e 选项**指定的值
  - **Entry(symbol)**指定的值
  - **.text 节**的起始地址
  - **入口点为 0**

常用的命令

包含其他 filename 的链接描述文件

INCULDE filename

指定多个输入文件名

INPUT(file,file,…)

常用的命令

指定输出文件的格式

OUTPUT_FORMAT(bfdname)

指定目标机器体系结构

OUTPUT_ARCH (bfdname)

例如：

OUTPUT_ARCH(arm)

常用的命令

- MEMORY：
  这个命令在用于嵌入式系统的链接描述文件中经常出现，它描述了各个内存块的起始地址和大小。格式如下：

```
MEMORY
{
    name [(attr)]:ORIGIN = origin,LENGTH =
len
    ...
}
```

例如：

```
MEMORY
{
    rom : ORIGIN=0x1000, LENGTH=0x1000
}
```

Memory 举例

```
//标注嵌入式设备中各个内存块的地址划分情况
MEMORY
{
  //标注flash中断向量表的起始地址为0x01000000,长度为0x0400
  romvec: ORIGIN  = 0x01000000, LENGTH  = 0x0400
  //标注flash的起始地址为0x01000400,长度为0x011fffff-0x01000400
  flash: ORIGIN = 0x01000400, LENGTH = 0x011fffff-0x01000400
  //标注flash的结束地址在0x011fffff
  eflash : ORIGIN = 0x011fffff, LENGTH = 1
  ramvec:ORIGIN = 0x00000000, LENGTH = 0x0400
  ram : ORIGIN = 0x00000400, LENGTH = 0x0003ffff-0x00000400
  eram : ORIGIN = 0x0003ffff, LENGTH = 1
}
```

SECTIONS 命令 I

- SECTIONS
  告诉 ld 如何把输入文件的各个节映射到输出文件的各个节中。
  - 在一个链接描述文件中**只能有一个 SECTIONS**命令
- 在 SECTIONS 命令中可以使用的命令有三种：
  - 定义入口点
  - 赋值
  - 定义输出节

SECTIONS 命令 II

定义输出节

```
SECTIONS
{
    ...
    secname :
    {
        contents
    }
    ...
}
```

例如：

SECTIONS 命令 III

```
SECTIONS
{
        ROM:{*(.text)}>rom
}
```

定位计数器

- 定位计数器，The Location Counter
  - 一个特殊的 ld 变量，使用 “.” 表示
  - 总是在 SECTIONS 中使用

例如：

```
SECTIONS
{
    output:
    {
        file1(.text);
        . = . + 1000;
        file2(.text);
        . = . + 1000;
        file3(.text);
    } = 0x1234;
}
```

一个简单例子

- 下面是一个简单的例子：
  例中，输出文件包含 text，data，bss 三个节，而输入文件也只包含这 3 个节：

```
SECTIONS
{
    .=0x01000000;
    .text:{*(.text)};
    .=0x08000000;
    .data:{*(.data)};
    .bss:{*(.bss)};
}
```

SECTIONS 举例（对应于上面的 MEMORY 例子）

```
SECTIONS
{
  //定义输出文件的ramvec节
  .ramvec:
  {
  //设定一个变量_ramvec来代表当前的位置，即ramvec节的开始处
  _ramvec=.;
  }>ramvec     //把该节定义到MEMORY中ramvec所代表的内存块中

  //定义输出文件的data节
  .data:
  {
  //设定一个变量_data_start来代表当前的位置，即data节的开始处
  _data_start = .;
  //把所有输入文件中的.data节的数据放在此处
  *(.data)
  //设定一个变量_edata为.data节的结束地址
  _edata = .;
  //把_edata按16位对齐
  _edata = ALIGN(0x10);
  }>ram //把.data节的内容放到ram定义的MEMORY中

  ......
```

小结

前言

GNU Tools 简介

- GCC
- GNU binutils
- Gdb—调试器
- GNU make——软件工程工具
- GNU ld——链接器

小结和作业

作业：

1. 在课件中

Thanks !

The end.