Linux软件开发工具

本章内容

- Linux程序设计
- 论述如何使用Linux的C语言工具完成代码编辑，缩进，编译，模块化程序的处理，程序库的建立，代码管理及版本控制等工作

软件开发工具概述

概述

- Linux软件开发一直在Internet环境下进行。
- 大多数Linux软件是经过自由软件基金会（Free Software Foundation）提供的GNU（GNU 即 GNU's not UNIX）公开认证授权的，因而通常被称作GNU软件。

Linux程序设计语言

- 高级语言
  - C、C++、JAVA、FROTRAN、BASIC、LISP…
- 脚本语言（Script Language）
  - Shell: bash、sh、tcsh、 ksh
  - Python、PHP、JSP、JavaScript、Perl…

TIOBE编程语言排行榜

Nov 2020	Nov 2019	Change	Programming Language	Ratings	Change
1	2		C	16.21%	+0.17%
2	3		Python	12.12%	+2.27%
3	1		Java	11.68%	-4.57%
4	4		C++	7.60%	+1.99%
5	5		C#	4.67%	+0.36%
6	6		Visual Basic	4.01%	-0.22%
7	7		JavaScript	2.03%	+0.10%
8	8		PHP	1.79%	+0.07%
9	16		R	1.64%	+0.66%
10	9		SQL		
44			Bash	0.31%	

编程工具

- 命令行环境：
  - 编辑工具
    - vi、emacs、gedit、nano
  - 编译、链接
    - gcc
  - 调试
    - gdb
  - make命令
  - 版本控制工具
    - git、CVS等
- IDE：
  - Emacs/xemacs
  - Visual Studio Code
  - Eclipse
  - ……

Linux编程风格

Linux编程风格

1、GNU风格

- 函数返回类型说明和函数名分两行放置，函数起始字符和函数开头左花括号放到最左边。
- 尽量不要让两个不同优先级的操作符出现在相同的对齐方式中，应该附加额外的括号使得代码缩进可以表示出嵌套。
- 每个程序都应该以一段简短的说明其功能的注释开头。
- 请为每个函数书写注释，说明函数是做什么的，需要哪些入口参数，参数可能值的含义和用途。如果用了非常见的、非标准的东西，或者可能导致函数不能工作的任何可能的值，应该进行特殊说明。如果存在重要的返回值，也需要说明。
- 不要声明多个变量时跨行，每一行都以一个新的声明开头。
- 当一个if中嵌套了另一个if-else时，应用花括号把if-else括起来。
- 要在同一个声明中同时说明结构标识和变量或者结构标识和类型定义(typedef)。先定义变量，再使用。

GNU风格

- 尽量避免在if的条件中进行赋值。
- 在名字中使用下划线以分割单词，尽量使用小写；把大写字母留给宏和枚举常量，以及根据统一惯例使用的前缀。例如，应该使用类似ignore_space_change_flag的名字；不要使用类似iCantReadThis的名字。
- 用于表明一个命令行选项是否给出的变量应该在选项含义的说明之后，而不是选项字符之后被命名。

Linux编程风格

2. Linux 内核编程风格

- Linux内核缩进风格是8个字符。
- Linux内核风格采用K&R标准，将开始的大括号放在一行的最后，而将结束的大括号放在一行的第一位。
- 命名尽量简洁。不应该使用诸如ThisVariableIsATemporaryCounter之类的名字。应该命名为tmp，这样容易书写，也不难理解。但是命名全局变量，就应该用描述性命名方式，例如应该命名“count_active_users()”，而不是“cntusr()”。本地变量应该避免过长。
- 函数最好短小精悍，一般来说不要让函数的参数多于１０个，否则应该尝试分解这个过于复杂的函数。
- 通常情况，注释说明代码的功能，而不是其实现原理。避免把注释插到函数体内，而写到函数前面，说明其功能，如果这个函数的确很复杂，其中需要有部分注释，可以写些简短的注释来说明那些重要的部分，但是不能过多。

Linux内核使用GNU C和AT&T汇编。要了解linux编程风格

程序生成工具

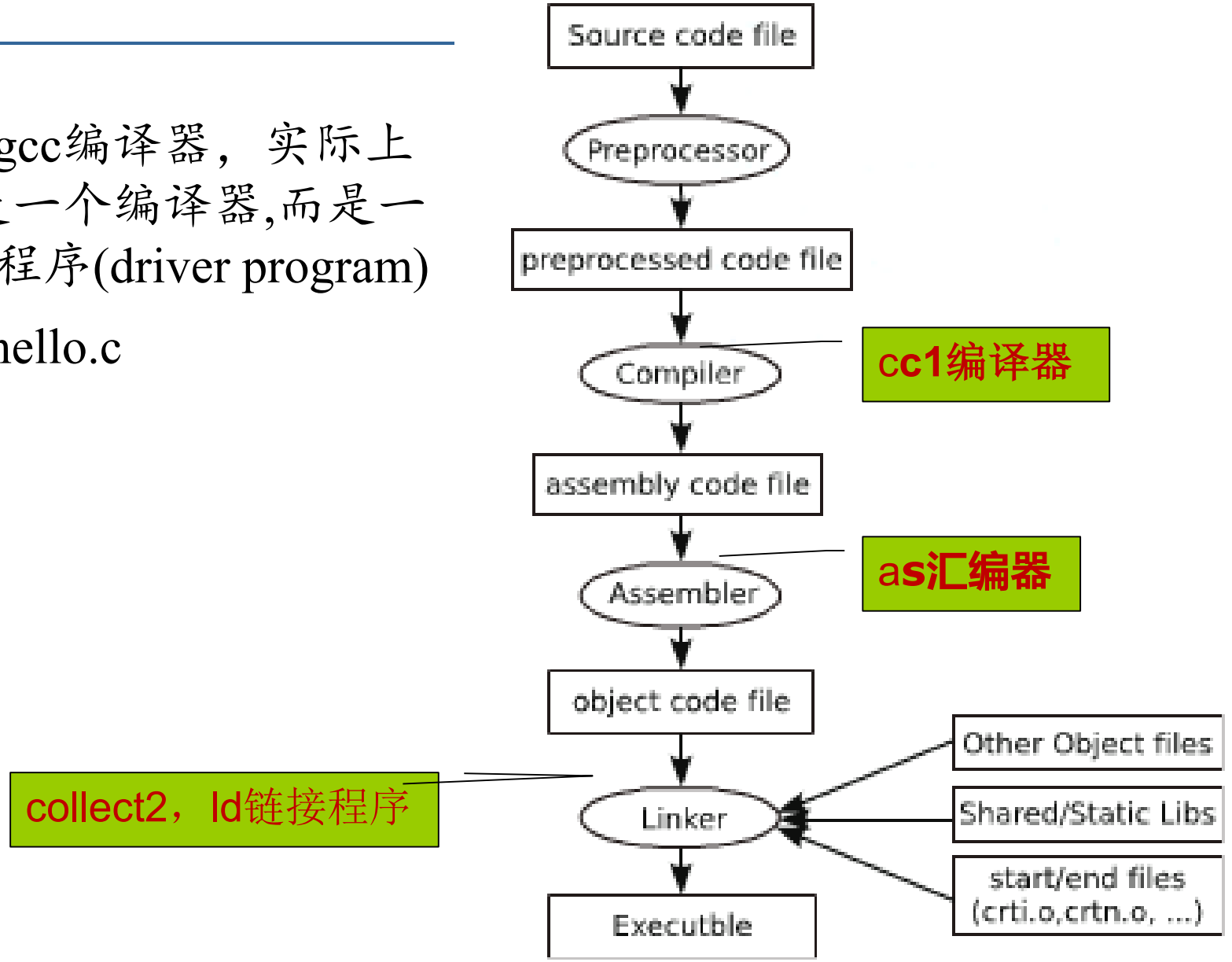
- 生成c语言源代码：vi、emacs、gedit、kedit等编辑器
- 缩进C语言代码
  - 最著名的缩进风格是Brian Kernighan和Dennis Ritchie(Unix和C语言的发明者)在世界上第一本讲C语言的书——《C程序设计语言》中建议使用的(即所谓的K&R风格)。
  - LINUX提供了一个indent命令用于自动调整C代码的缩进风格，
  - 默认情况下indent会按照GNU风格进行缩进，并保留用户输入的所有回车符。

编译器

c程序的编译

- Linux最常用的c编译器是gcc（GNU c/c++ compiler）
- GNU CC（简称为gcc，GNU compiler collection）是GNU项目中符合ANSI C标准的编译器集合，能够编译用C、C++和Object C等语言编写的程序。
- gcc不仅功能强大，而且可以编译如C、C++、Object C、Java、Fortran、Pascal、Modula-3和Ada等多种语言，而且gcc又是一个交叉平台编译器，它能够在当前CPU平台上为多种不同体系结构的硬件平台开发软件，因此尤其适合在嵌入式领域的开发编译。

- 通常称gcc编译器，实际上gcc不是一个编译器,而是一个驱动程序(driver program)
- gcc –v hello.c



gcc所支持后缀名解释

后　缀　名	所对应的语言	后　缀　名	所对应的语言
.c	C 原始程序	.s/.S	汇编语言原始程序
.C/.cc/.cxx	C++原始程序	.h	预处理文件（头文件）
.m	Objective-C 原始程序	.o	目标文件
.i	已经过预处理的 C 原始程序	.a/.so	编译后的库文件
.ii	已经过预处理的 C++原始程序		

gcc --- 常用选项

选项	含义
-c	只编译不链接，生成目标文件“.o”
-S	只编译不汇编，生成汇编代码
-E	只进行预编译，不做其他处理
-g	在可执行程序中包含标准调试信息
-o file	指定将 file 文件作为输出文件
-v	打印出编译器内部编译各过程的命令行信息和编译器的版本
-I dir	在头文件的搜索路径列表中添加 dir 目录

gcc ---库选项

选　　项	含　　义
-static	进行静态编译，即链接静态库，禁止使用动态库
-shared	1．可以生成动态库文件 2．进行动态编译，尽可能地链接动态库，只有没有动态库时才会链接同名的静态库（默认选项，即可省略）
-L dir	在库文件的搜索路径列表中添加 dir 目录
-lname 如：-lpthread	链接称为 libname.a（静态库）或者 libname.so（动态库）的库文件。若两个库都存在，则根据编译方式（-static 还是-shared）而进行链接
-fPIC（或-fpic）	生成使用相对地址的位置无关的目标代码（Position Independent Code）。然后通常使用 gcc 的-static 选项从该 PIC 目标文件生成动态库文件。

gcc

- gcc命令
  - 语法：gcc [选项] 文件列表
  - 常用选项/功能：

    -c　　编译成目标（.o）文件

    -l 库文件名　连接库文件

    -o 文件名　将生成的可执行文件保存到指定文件中，默认是a.out

C程序的编译

- mypro1.c文件：

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

- 利用如下的命令可编译生成可执行文件：

```
$ gcc -o mypro1 mypro1.c
```

- 生成了mypro1可执行文件，运行这个程序输入

```
$ ./mypro1
Hello  world!
```

- 如果没有-o选项，则生成a.out执行文件

```
$ gcc mypro1.c
$ ./a.out
```

头文件

- GCC编译器缺省的头文件目录是/usr/include目录及其子目录下。
- 那些依赖于特定 Linux版本的头文件通常可在目录/usr/include/sys和/usr/include/linux中找到。
- 其他编程系统也有各自的include文件，并将其存储在可被相应编译器自动搜索到的目录里。例如，X视窗系统的/usr/include/X11目录和GNU C++的/usr/include/c++目录。
- 调用C语言编译器时，我们可以使用-I标志来包含保存在子目录或非标准位置中的include文件。例如：

  gcc –I /usr/openwin/include power.c

  - 它指示编译器不仅在标准位置，也在/usr/openwin/include目录中查找程序power.c中包含的头文件。

库文件

- 使用gcc的-l选项可以连接已有的程序库
  - 数学库libm.a连接到power.o

gcc –o power power.o -lm

gcc –o power power.o /usr/lib/libm.a

a代表传统的静态函数库；so代表共享函数库。

文件名中“lib”以后，扩展名以前的部分

- 请参看C语言编译器的使用手册（man gcc）以了解更多细节。

glibc函数库

- GNU的C函数库glibc定义了ISO C标准指定的所有库函数，以及由POSIX或其他UNIX类操作系统指定附加特色，还包括与GNU系统相关的扩展。glibc基于如下标准：
  - ISO C：C语言的国际标准，即ANSI C
  - POSIX： glibc实现了POSIX的API即POSIX.1所指定的所有函数，该标准是对ISO C的扩展
  - Berkeley UNIX(BSD和SunOS)
  - SVID（System V的接口描述）
  - XPG（X/Open可移植性指南）

make

make程序

- make程序：是一个命令工具，是一个解释makefile中指令的命令工具
- make程序提供一种可以用于构件大规模工程的、强劲的而灵活的机制。

句法：	make［选项］［目标］［宏定义］
用途：	make工具根据名为makefile或Makefile的文件中指定的依赖关系对系统进行更新。［选项］［目标］［宏定义］可以以任意顺序指定。
常用选项/特性：	-d 显示调试信息 -f 文件 此选项告诉make使用指定文件作为依赖关系文件，而不是默认的makefile或Makefile，如果指定的文件名是"-"，那么make将从标准输入读入依赖关系。 -h 显示所有选项的帮助信息 -n 测试模式，并不真的执行任何命令，只是显示输出这些命令 -s 安静模式--不输出任何提示信息。

make程序

- make工具依赖一个特殊的，名为makefile的文件，这个文件描述了系统中各个模块之间的依赖关系。
- GNU make的主要功能是读进一个文本文件makefile并根据makefile的内容执行一系列的工作。
- makefile的默认文件名为GNUmakefile、makefile或Makefile，当然也可以在make的命令行中指定别的文件名。如果不特别指定，make命令在执行时将按顺序查找默认的makefile文件。
- 多数Linux程序员使用第三种文件名Makefile。因为第一个字母是大写，通常被列在一个目录的文件列表的最前面。

make程序

- Makefile是一个文本形式的数据库文件，其中包含一些规则来告诉make处理哪些文件以及如何处理这些文件。这些规则主要是描述哪些文件（称为target目标文件，不要和编译时产生的目标文件相混淆）是从哪些别的文件（称为dependency依赖文件）中产生的，以及用什么命令（command）来执行这个过程。
- make会对磁盘上的文件进行检查，如果目标文件的生成或被改动时的时间（称为该文件时间戳）至少比它的一个依赖文件还旧的话，make就执行相应的命令，以更新目标文件。目标文件不一定是最后的可执行文件，可以是任何一个中间文件并可以作为其他目标文件的依赖文件。

make程序

- Makefile规则的语法格式：

  目标文件列表：依赖文件列表

  <tab >命令列表

- 一个Makefile文件主要含有一系列的规则，每条规则包含以下内容。
  - “目标文件列表”，即make最终需要创建的文件，如可执行文件和目标文件；目标也可以是要执行的动作，如“clean”。
  - “依赖文件列表”，通常是编译目标文件所需要的其他文件。
  - “命令列表”，是make执行的动作，通常是把指定的相关文件编译成目标文件的编译命令，每个命令占一行，且每个命令行的起始字符必须为TAB字符。
- 除非特别指定，否则make的工作目录就是当前目录。“目标文件列表”是需要创建的二进制文件或目标文件，依赖文件列表是在创建“目标文件列表”时需要用到的一个或多个文件的列表，命令序列是创建“目标文件列表”文件所需要执行的步骤，比如编译命令。

make程序

- 例如，有以下的Makefile文件：

```
# 一个简单的Makefile的例子
test：prog.o code.o
        gcc –o test prog.o code.o
prog.o：prog.c prog.h code.h
        gcc –c prog.c –o prog.o
code.o：code.c code.h
        gcc –c code.c –o code.o
clean：
        rm –f *.o
```

make程序

- 上面的Makefile文件中共定义了四个目标：test、prog.o、code.o和clean。目标从每行的最左边开始写，后面跟一个冒号（：），如果有与这个目标有依赖性的其他目标或文件，把它们列在冒号后面，并以空格隔开。然后另起一行开始写实现这个目标的一组命令。
- 在Makefile中，可使用续行号（\）将一个单独的命令行延续成几行。但要注意在续行号（\）后面不能跟任何字符（包括空格和键）。
- 一般情况下，调用make命令可输入：$make target
- target是Makefile文件中定义的目标之一，如果省略target，make就将生成Makefile文件中定义的第一个目标。对于上面Makefile的例子，单独的一个“make”命令等价于：

  $ make test
- 因为test是Makefile文件中定义的第一个目标，make首先将其读入，然后从第一行开始执行，把第一个目标test作为它的最终目标，所有后面的目标的更新都会影响到test的更新。第一条规则说明只要文件test的时间戳比文件prog.o或code.o中的任何一个旧，下一行的编译命令将会被执行。

make程序

- 在检查文件prog.o和code.o的时间戳之前，make会在下面的行中寻找以prog.o和code.o为目标的规则，在第三行中找到了关于prog.o的规则，该文件的依赖文件是prog.c、prog.h和code.h。
- 同样，make会在后面的规则行中继续查找这些依赖文件的规则，如果找不到，则开始检查这些依赖文件的时间戳，如果这些文件中任何一个的时间戳比prog.o的新，make将执行“gcc –c prog.c –o prog.o”命令，更新prog.o文件。
- 以同样的方法，接下来对文件code.o做类似的检查，依赖文件是code.c和code.h。当make执行完所有这些套嵌的规则后，make将处理最顶层的test规则。如果关于prog.o和code.o的两个规则中的任何一个被执行，至少其中一个.o目标文件就会比test新，那么就要执行test规则中的命令，因此make去执行gcc命令将prog.o和code.o连接成目标文件test。

make程序

- 在上面Makefile的例子中，还定义了一个目标clean，它是Makefile中常用的一种专用目标，即删除所有的目标模块。
- make做的工作：
  - 首先make按顺序读取makefile中的规则，
  - 然后检查该规则中的依赖文件与目标文件的时间戳哪个更新，如果目标文件的时间戳比依赖文件还早，就按规则中定义的命令更新目标文件。如果该规则中的依赖文件又是其他规则中的目标文件，那么依照规则链不断执行这个过程，直到Makefile文件的结束，至少可以找到一个不是规则生成的最终依赖文件，获得此文件的时间戳，然后从下到上依照规则链执行目标文件的时间戳比此文件时间戳旧的规则，直到最顶层的规则。

Makefile中的变量（宏）

- Makefile里的变量就像一个环境变量。事实上，环境变量在make中也被解释成make的变量。这些变量对大小写敏感，一般使用大写字母。几乎可以从任何地方引用定义的变量，变量的主要作用如下：
  - 保存文件名列表。在前面的例子里，作为依赖文件的一些目标文件名出现在可执行文件的规则中，而在这个规则的命令行里同样包含这些文件并传递给gcc做为命令参数。如果使用一个变量来保存所有的目标文件名，则可以方便地加入新的目标文件而且不易出错。
  - 保存可执行命令名，如编译器。在不同的Linux系统中存在着很多相似的编译器系统，这些系统在某些地方会有细微的差别，如果项目被用在一个非gcc的系统里，则必须将所有出现编译器名的地方改成用新的编译器名。但是如果使用一个变量来代替编译器名，那么只需要改变该变量的值。其他所有地方的命令名就都改变了。
  - 保存编译器的参数。在很多源代码编译时，gcc需要很长的参数选项，在很多情况下，所有的编译命令使用一组相同的选项，如果把这组选项使用一个变量代表，那么可以把这个变量放在所有引用编译器的地方。当要改变选项的时候，只需改变一次这个变量的内容即可。

Makefile中的变量

- Makefile中的变量是用一个文本串在Makefile中定义的，这个文本串就是变量的值。只要在一行的开始写下这个变量的名字，后面跟一个“＝”号，以及要设定这个变量的值即可定义变量，下面是定义变量的语法：

  VARNAME=string

- 使用时，把变量用括号括起来，并在前面加上$符号，就可以引用变量的值：

  $(VARNAME)

- make解释规则时，VARNAME在等式右端展开为定义它的字符串。变量一般都在Makefile的头部定义。按照惯例，所有的Makefile变量名都应该是大写。如果变量的值发生变化，就只需要在一个地方修改，从而简化了Makefile的维护。

Makefile中的变量

- 现在利用变量把前面的Makefile重写一遍：

```
OBJS=prog.o code.o
CC=gcc
 test：$(OBJS)
      $(CC) –o test $(OBJS)
 prog.o：prog.c prog.h code.h
      $(CC) –c prog.c –o prog.o
 code.o：code.c code.h
      $(CC) –c code.c –o code.o
 clean：
      rm –f *.o
```

Makefile中的变量

- 除用户自定义的变量外，make还允许使用环境变量、自动变量和预定义变量。
- 使用环境变量的方法很简单，在make启动时，make读取系统当前已定义的环境变量，并且创建与之同名同值的变量，因此用户可以像在shell中一样在Makefile中方便的引用环境变量。
- 需要注意，如果用户在Makefile中定义了同名的变量，用户自定义变量将覆盖同名的环境变量。
- Makefile中还有一些预定义变量和自动变量，但是看起来并不像自定义变量那样直观。

图 make关系树的第一层

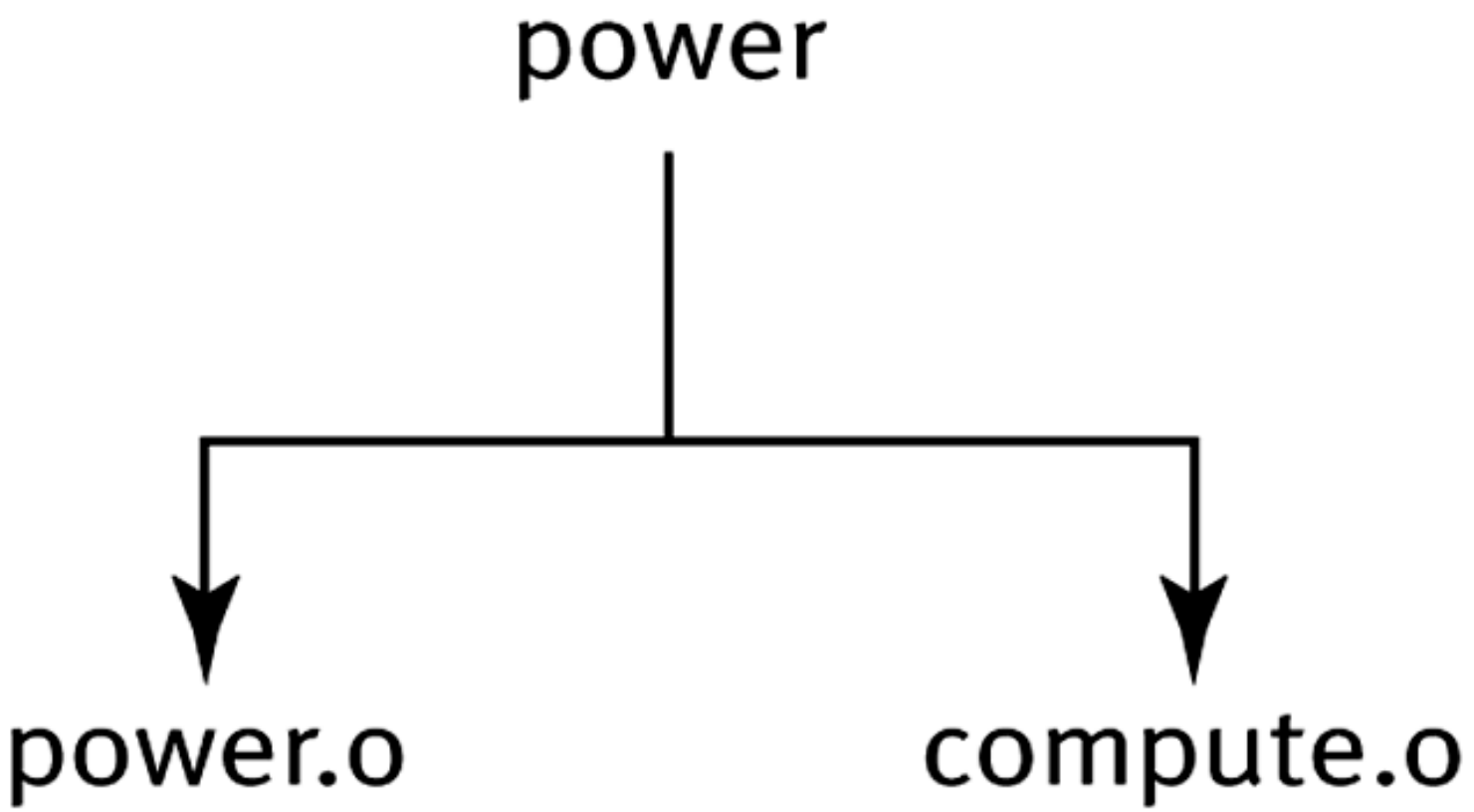


图 make关系树的前两层

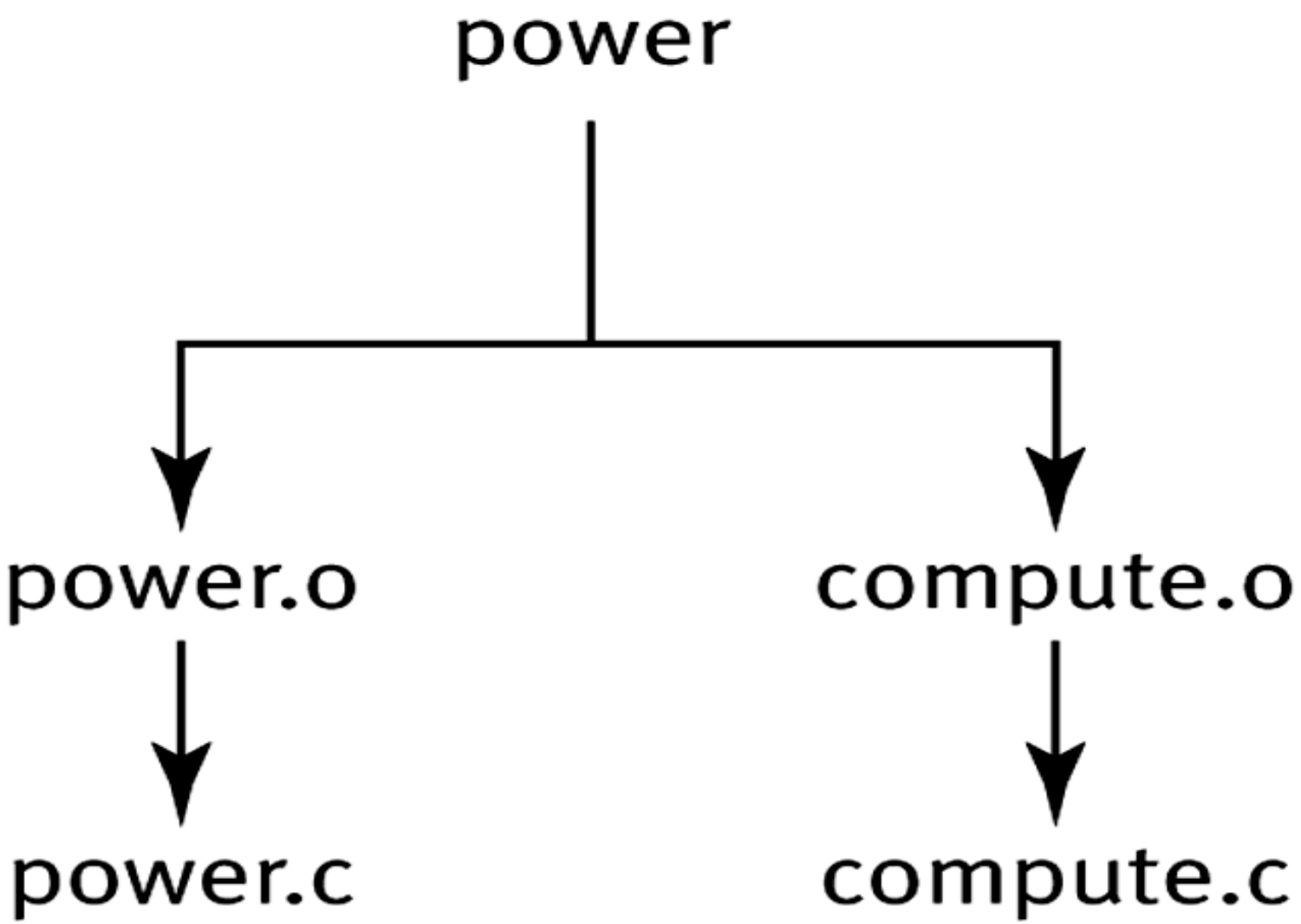
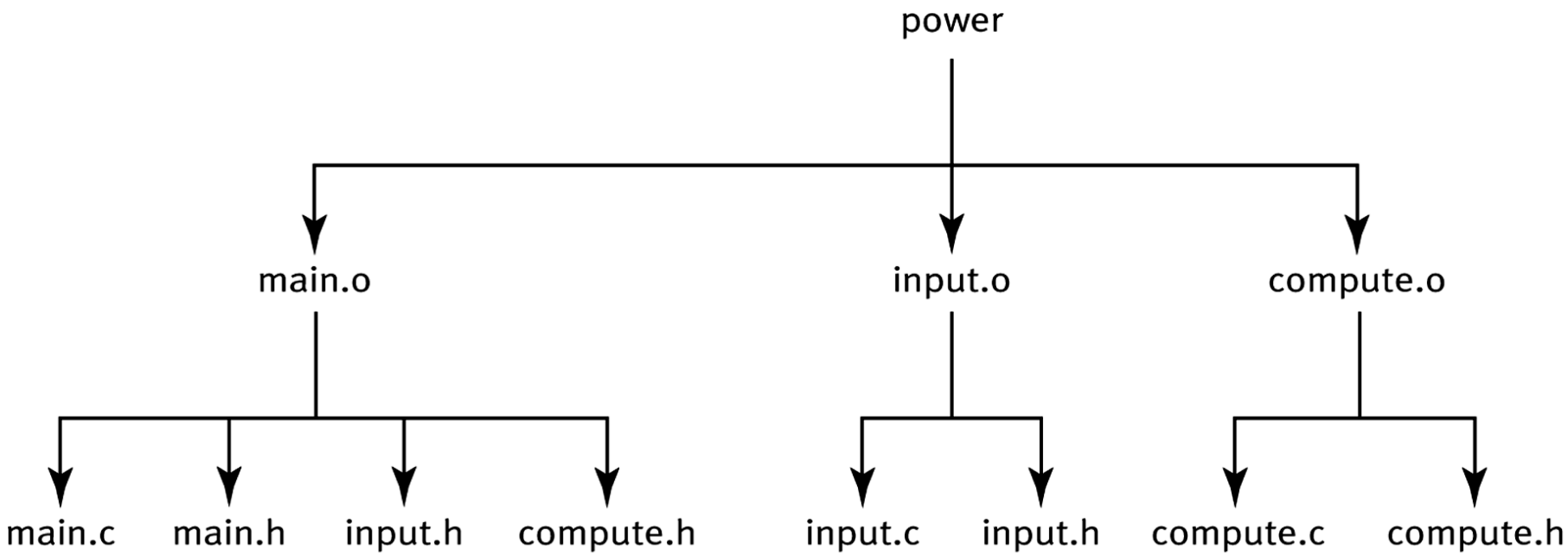


图 一个简单程序的make关系树



动态分析工具GDB

- Linux系统中包含了GNU 调试程序gdb，它是一个用来调试C和 C++ 程序的调试器。可以使程序开发者在程序运行时观察程序的内部结构和内存的使用情况。gdb 所提供的一些功能如下所示：
  - 运行程序，设置所有的能影响程序运行的参数和环境；
  - 控制程序在指定的条件下停止运行；
  - 当程序停止时，可以检查程序的状态；
  - 修改程序的错误，并重新运行程序；
  - 动态监视程序中变量的值；
  - 可以单步执行代码，观察程序的运行状态。

调试工具GDB

句法：	gdb [选项][可执行程序[core文件|进程ID]]
用途：	跟踪指定程序的运行，给出它的内部运行状态以协助你定位程序的bug。你还可以指定一个程序运行错误产生的core文件，或者正在运行的程序进程ID.
常用选项/特性：	-c core文件　　使用指定core文件检查程序 -h　　列出命令行选项的简要介绍 -n　　忽略~/.gdbinit文件中指定的执行命令 -q　　禁止显示介绍信息和版权信息 -s 文件　　使用保存在指定文件中的符号表

调试工具GDB

- gdb程序调试的对象是可执行文件，而不是程序的源代码文件。然而，并不是所有的可执行文件都可以用gdb调试。如果要让产生的可执行文件可以用来调试，需在执行gcc指令编译程序时，加上-g参数，指定程序在编译时包含调试信息。
- 调试信息包含程序里的每个变量的类型和在可执行文件里的地址映射以及源代码的行号。gdb 利用这些信息使源代码和机器码相关联。
- 在命令行上输入gdb并按回车键就可以运行gdb了，如果一切正常的话，将启动gdb，可以在屏幕上看到以下的内容：

调试工具GDB

GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or 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.
This GDB was configured as "i386-redhat-linux-gnu".
(gdb)

- 启动gdb后，可以在命令行上指定很多的选项。输入：help.可以获得gdb的帮助信息。如果想要了解某个具体命令（比如break）的帮助信息，在gdb提示符下输入下面的命令：break屏幕上会显示关于break的帮助信息。从返回的信息可知，break是用于设置断点的命令。
- 在Linux Shell提示符输入：man gdb 可以看到man的手册页。

调试工具GDB

- gdb支持很多的命令且能实现不同的功能。这些命令从简单的文件装入到允许你检查所调用的堆栈内容的复杂命令，下面列出了在使用gdb 调试时会用到的一些命令。
  - 1）file命令：装入想要调试的可执行文件。
  - 2）cd命令：改变工作目录。
  - 3）pwd命令：返回当前工作目录。
  - 4）run命令：执行当前被调试的程序。
  - 5）kill命令：停止正在调试的应用程序。
  - 6）list命令：列出正在调试的应用程序的源代码。
  - 7）break命令：设置断点。
  - 8）tbreak命令；设置临时断点。它的语法与break相同。区别在于用tbreak设置的断点执行一次之后立即消失。

调试工具GDB

- 9）watch命令：设置监视点，监视表达式的变化。
- 10）awatch命令：设置读写监视点。当要监视的表达式被读或写时将应用程序挂起。它的语法与watch命令相同。
- 11）rwatch命令：设置读监视点，当监视表达式被读时将程序挂起，等待调试。此命令的语法与watch相同。
- 12）next命令：执行下一条源代码，但是不进入函数内部。也就是说，将一条函数调用作为一条语句执行。执行这个命令的前提是已经run，开始了代码的执行。
- 13）step命令：执行下一条源代码，进入函数内部。如果调用了某个函数，会跳到函数所在的代码中等候一步步执行。执行这个命令的前提是已经用run开始执行代码。
- 14）display命令：在应用程序每次停止运行时显示表达式的值。
- 15）info break命令：显示当前断点列表，包括每个断点到达的次数。

调试工具GDB

- 16）info files命令：显示调试文件的信息。
- 17）info func命令：显示所有的函数名。
- 18）info local命令：显示当前函数的所有局部变量的信息。
- 19）info prog命令：显示调试程序的执行状态。
- 20）print命令；显示表达式的值。
- 21）delete命令：删除断点。指定一个断点号码，则删除指定断点。不指定参数则删除所有的断点。
- 22）shell命令：执行Linux Shell命令。
- 23）make命令：不退出gdb而重新编译生成可执行文件。
- 24） help命令：列出gdb命令的种类
- 25） quit命令：退出gdb。

调试工具GDB

- $ gcc -g bugged.c -o bugged
- $ gdb -q bugged

  (gdb)
- 进入gdb环境以后，你可以使用很多命令监视欲调试程序的运行。

运行效率分析

- time命令用于对一个程序或任何shell命令的运行效率分析。
- 此命令计算三个时间：总时间、系统时间和用户时间，并按“时:分:秒.百分之一秒”的格式输出。
- 句法：

  time [用户命令]

  - 用途：报告命令的三种执行时间以便用户进行运行效率分析。三个时间分别是总时间(程序从开始运行到完成的时间)、系统时间(系统干预活动占用的时间)和用户时间(程序代码执行的时间)。

End of chapter