第 3 章　Linux 系统编程基础

本章首先介绍 GCC 编译器的编译过程及常用选项的使用，通过实例介绍 GDB 调试器的使用方法，然后介绍 Make 工具的使用，最后介绍文件操作、时间获取和多线程等任务的编程方法。

3.1　GCC 编译器

3.1.1　GCC 概述

GCC(GNU C Compiler)是 GUN 项目的 C 编译套件，也是 GNU 软件家族中的代表产品之一。GCC 目前支持的体系结构有四十余种，如 X86、ARM、PowerPC 等系列处理器；能运行在不同的操作系统上，如 Linux、Android、Solaris、Windows CE 等操作系统；可完成 C、C++、Objective C 等源文件向运行在特定 CPU 硬件上的目标代码的转换。GCC 的执行效率与一般的编译器相比平均效率要高 20%～30%。GCC 是 Linux 平台下最常用的编译器之一，它也是 Linux 平台编译器事实上的标准。同时，在使用 Linux 操作系统的嵌入式开发领域，GCC 也是使用最普遍的编译器之一。

GCC 编译器与 GUN Binutils 工具包是紧密集成的，如果没有 Binutils 工具，GCC 也不能正常工作。Binutils 是一系列开发工具，包括连接器、汇编器和其他用于目标文件和档案的工具。Binutils 工具集里主要包括一系列程序，如 addr2line、ar、as、C++、gprof、ld、nm、objcopy、objdump、ranlib、readelf、size、strings 和 strip 等，它包括的库文件有 libiberty. a、libbfd. a、libbfd. so、libopcodes. a 和 libopcodes. so 等。

在 Linux 操作系统中，文件名的扩展名不代表文件的类型，但为了提高工作效率，通常会给每种文件定义一个扩展名。GCC 支持的文件类型比较多，具体如表 3.1 所示。

表 3.1　GCC 支持的文件类型

扩展名	说　明	扩展名	说　明
. c	C 源程序	. ii	经过预处理的 C++程序
. a	由目标文件构成的档案文件(库文件)	. m	Objective C 源程序
. C、. cc	C++源程序	. o	编译后的目标程序
. h	头文件	. s	汇编语言源程序
. i	经过预处理的 C 程序	. S	经过预编译的汇编程序

3.1.2　GCC 编译过程

1. 编译示例

这里通过一个常用的例子来说明 GCC 的编译过程。

(1) 利用文本编辑器创建 hello.c 文件,程序内容如下。

```
#include<stdio.h>
void main()
{
    char msg[80]="Hello,world!";
    printf("%s\n",msg);
}
```

(2) 编写完后,执行编译指令。

```
#gcc hello.c
```

因为编译时没有加任何选项,所以会默认生成一个名为 a.out 的可执行文件。执行该文件的命令及结果如下。

```
#./a.out
    Hello,world!
```

注意:在 Linux 系统中,./表示在当前目标下执行程序。

2. 编译过程

使用 GCC 由 C 语言源代码程序生成可执行文件要经历四个过程,如图 3.1 所示。

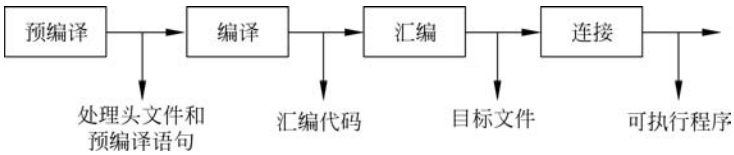

图 3.1　GCC 编译过程

1) 预编译

预编译(Preprocessing)过程的主要功能是读取源程序,并对头文件(include)、预编译语句(如 define 等)和一些特殊符号进行分析和处理。如把头文件复制到源文件中,并将输出的内容送到系统的标准输出。源代码中的预编译指示以"#"为前缀。通过在 gcc 后加上-E 选项完成对代码的预编译。预编译命令如下。

```
# gcc -E hello.c
```

执行命令时,控制台上会有数千行的输出,其中大多数来自 stdio.h 头文件,也有部分是声明。预编译主要完成 3 个具体任务,包括把 include 中的头文件复制到要编译的源文件中、用实际值替代 define 文本、在调用宏的地方进行宏替换。

下面通过实例 test.c 理解预编译完成的工作任务,test.c 的代码如下。

```
#define number 1+2
int main()
{
    int n;
    n=number*3;
    return 0;
}
```

对 test.c 文件进行预编译,需要输入以下命令。

```
#gcc -E test.c
```

执行命令后会显示如下内容。

```
# 1 "test.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "test.c"

int main()
{
 int n;
 n=1+2*3;
 return 0;
}
```

如果要将预编译结果保存在 test.i 文件中,可以输入以下命令。

```
#gcc -E test.c -o test.i
```

2) 编译

编译(Compilation)的主要功能包括两部分,第一部分是检查代码的语法,如果出现语法错误,则给出错误提示代码,并结束编译,只有在代码无语法错误的情况下才能进入第二部分;第二部分是将预编译后的文件转换成汇编语言,并自动生成扩展名为.s 的文件。编译的命令如下。

```
#gcc -S test.c
```

执行命令后会生成一个名为 test.s 的汇编程序,文件内容如下。

```
    .file   "test.c"
    .text
.globl main
    .typemain,@function
main:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    movq    %rsp,%rbp
    .cfi_offset 6,-16
    .cfi_def_cfa_register 6
    movl    $7,-4(%rbp)
    movl    $0,%eax
    leave
    ret
    .cfi_endproc
.LFE0:
    .sizemain,.-main
```

```
    .ident    "GCC: (Ubuntu/Linaro 4.4.7-1ubuntu2) 4.4.7"
    .section    .note.GNU-stack,"",@progbits
```

3）汇编

汇编(Assembly)的主要功能是将汇编语言代码变成目标代码(机器代码)。汇编只是将汇编语言代码转换成目标代码,但不进行连接,目标代码不能在 CPU 上运行。汇编使用选项为-c,它会自动生成一个扩展名为.o 的目标程序。汇编的命令如下。

```
#gcc -c test.c
```

执行命令后会生成一个名为 test.o 的目标文件,目标文件是一个二进制文件,所以不能用文本编辑器来查看它的内容。

4）连接

连接(Linking)的主要功能是连接目标代码,并生成可执行文件。连接的命令如下。

```
#gcc test.o -o test
```

也可以利用如下命令执行连接过程。

```
#gcc test.c -o test
```

执行命令后会生成一个名为 test 的可执行文件。通过执行./test 命令,就可以运行指定的程序。

3.1.3　GCC 选项

GCC 编译器提供了较多的选项,选项必须以"-"开始,常用的选项如表 3.2 所示。

表 3.2　GCC 常用选项

选　　项	说　　明
-c	只编译生成目标文件,扩展名为.o
-E	只进行预编译,不做其他处理
-g	在执行程序中包括标准调试信息
-I DirName	将 DirName 加入头文件的搜索目录列表中
-L DirName	将 DirName 加入库文件的搜索目录列表中,在默认情况下 gcc 只链接共享库
-l FOO	链接名为 libFOO 的函数库
-O	整个源代码会在编译、连接过程中进行优化处理,可执行文件的执行效率可以提高,但是编译、连接的速度就相应慢些
-O2	比-O 更好的优化,但编译链接速度更慢
-o FileName	指定输出文件名,如果没有指定,默认文件名是 a.out
-pipe	在编译过程的不同阶段间使用管道
-S	只编译不汇编,生成汇编代码
-static	链接静态库
-Wall	指定产生全部的警告信息

1. 输出文件选项

如果不使用任何选项进行编译,生成的可执行文件都是 a.out。如果要指定输出的文

件名，可以使用选项-o。例如将源文件 hello.c 编译成可执行文件 hello，命令格式如下。

```
#gcc hello.c -o hello
```

2. 链接库文件选项

Linux 操作系统下的库文件包括两种格式：一种是动态链接库；另一种是静态链接库。动态链接库的扩展名为.so，静态链接库的扩展名为.a。动态链接库是在程序运行过程中进行动态加载，静态链接库是在编译过程中完成静态加载。

使用 GCC 编译时，编译器会自动调用 C 标准库文件，但当要使用标准库以外的库文件时，一定要使用选项-l 来指定具体库的文件名，否则会报编译错误，如报 undefined reference to ‘xxxx’错误。Linux 操作系统下的库文件都是以 lib 开头，因此在使用-l 选项指定链接的库文件名时可以省去 lib。

例如一个多线程程序 pthread.c，代码如下。

```
#include <stdio.h>
#include "pthread.h"
void * producer(void * data)
{
 printf("producer end!\n");
  return NULL;
}
int main(void)
{
        pthread_t th_a;
        void * retval;
        pthread_create(&th_a,NULL,producer,0);
        pthread_join(th_a,&retval);
        return 0;
}
```

将 pthread.c 编译成名为 pthread 的可执行程序，如果输入以下命令，则会出错。

```
#gcc pthread.c -o pthread
/tmp/ccQEeIpc.o: In function 'main':
pthread.c:(.text+0x3c): undefined reference to 'pthread_create'
pthread.c:(.text+0x4f): undefined reference to 'pthread_join'
collect2: ld 返回 1
```

以上编译提示信息的意思是在主函数中有 pthread_create 和 pthread_join 两个函数，但在进行链接时，函数库中没有找到这两个函数。因为在标准库文件中的确没有这两个函数，它们在多线程 libpthread.a 或 libpthread.so 库文件中，库文件保存在/usr/lib 目录下。如果只生成目标文件，而不链接生成可执行文件，则不需要指定库文件名，如输入以下命令。

```
#gcc pthread.c -c -o pthread.o
```

如果要链接生成可执行程序，必须指定库文件名，正确命令如下。

```
#gcc pthread.c -lpthread -o pthread
```

GCC 在默认情况下优先使用动态链接库，当需要强制使用静态链接库时，需要加上-static 选项。使用静态链接库编译生成一个名为 pthread-s 的可执行程序的命令如下。

```
#gcc pthread.c -static -lpthread -o pthread-s
```

可以使用 ls -l 命令比较文件的大小，会发现 pthread-s 比 pthread 文件大很多。

```
-rwxrwxr-x 1 linux linux 8546 3 月 15 16:56 pthread
-rwxrwxr-x 1 linux linux 1141448 3 月 15 16:56 pthread-s
```

3. 指定库文件目录选项

编译时，编译器会自动到默认目录（一般为/usr/lib）寻找库文件，但当编译时所用的库文件不在默认目录时，就需要使用-L 选项来指定库文件所在的目录。如果不指定库文件所在目录，编译时会报 cannot find lxxx 错误。

下面通过一个实例来帮助读者理解。首先编写一个 chang 函数实现字符大小写转换功能，把该函数加入 libnew.so 库文件，将库文件保存到/home/test/lib 目录，最后编写一个程序 my.c 来调用 libnew.so 库文件中的 chang 函数，具体步骤如下。

（1）新建 chang.c 文件，该文件只有 chang 函数，内容如下。

```
char chang(char ch)
{
    if(ch>='A' && ch<='Z')
        ch=ch+32;
    else if(ch>='a' && ch<='z')
        ch=ch-32;
    return ch;
}
```

（2）将源文件生成目标文件，然后将目标文件添加到库文件中，命令如下。

```
#gcc -c chang.c -o chang.o
#ar rcs libnew.so chang.o
```

执行完以上命令，会生成 libnew.so 库文件，将库文件复制到/home/test/lib 目录。

（3）新建程序 my.c，在程序中调用 chang 函数，程序内容如下。

```
#include<stdio.h>
#include<string.h>
char chang(char ch);            //它是 libnew.so 库文件中的函数
main()
{
    char s[]="abCD12";
    int i,n;
    printf("1-%s\n",s);
    n=strlen(s);
    for(i=0;i<n;i++)
        s[i]=chang(s[i]);  //调用 libnew.so 库文件中的函数
    printf("2-%s\n",s);
}
```

将程序 my.c 编译成可执行程序 my，命令如下。

```
#gcc my.c -L /home/test/lib -lnew -o my
```

4. 指定头文件目录选项

编译时，编译器会自动到默认目录（一般为/usr/include）寻找头文件，但当文件中的头文件不在默认目录下时，就需要使用-I 选项来指定头文件所在的目录。如果不指定头文件所在目录，编译时会报 xxx.h：No such file or directory 错误。

例如程序 someapp.c，代码如下。

```
#include<stdio.h>
#include<someapp.h>
main()
{
    float s,r=3;
    s=PI*r*r;
    printf("s=%6.2f\n",s);
}
```

程序中的头文件 someapp.h 保存在/home/test/include 目录下，someapp.h 内容如下。

```
#define PI 3.14
```

将 someapp.c 编译成可执行程序 someapp，命令如下。

```
#gcc someapp.c -I /home/test/include -o someapp
```

5. 警告选项

在编译过程中，编译器的警告信息对于程序员来说是非常重要的，GCC 包含完整的警告提示功能，以便确定代码是否正确，尽可能实现可移植性。GCC 的编译器警告信息选项如表 3.3 所示。

表 3.3　GCC 的警告选项

类　　型	说　　明
-Wall	启用所有警告信息
-Werror	在发生警告时取消编译操作，即将警告看作是错误
-w	禁用所有警告信息

下面通过实例介绍如何在编译时产生警告信息，代码如下。

```
#include<stdio.h>
int main ()
{
    int x,y;
    for(x=1;x<=5;x++)
    {
      printf("x=%d\n",x);
    }
}
```

使用以下命令进行编译。

```
#gcc example.c
```

编译过程中没有任何提示信息，直接生成一个 a.out 可执行文件。如果加入-Wall 选项进行编译，命令如下。

```
# gcc -Wall example.c -o example
```

编译过程将会出现下面的警告信息。

```
example.c : In function 'main'
example.c : 4 : warning : unused variable 'y'
example.c: 7 : warning : control reaches end of non-void function
```

第 1 条警告信息的意思是在 main 函数有警告信息。

第 2 条警告信息的意思是变量 y 在程序中未使用。

第 3 条警告信息的意思是 main 函数返回类型为 int，但在程序中没有 return 语句。

GCC 给出的警告从严格意义上不算错误，但是可能会成为错误的栖息之地。所以在进行嵌入式软件开发时，需要重视警告信息，最好根据警告信息对源程序进行修改，直至编译时没有任何警告信息。

-Werror 选项会要求 GCC 将所有警告信息当成错误进行处理，需要将所有警告信息都修改消除后才能生成可执行文件，命令如下。

```
#gcc -Wall -Werror example.c -o example
```

当需要忽略警告信息时，可以使用-w 选项，命令如下。

```
#gcc -w example.c -o example
```

6. 调试选项

代码通过编译并不代表能正常工作了，还应通过调试器检查代码，以便更好地找到程序中的问题。Linux 操作系统下主要采用 GDB 调试器，在使用 GDB 之前，执行程序中要包括标准调试信息，加入的方法是采用调试选项-g，具体命令如下。

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

7. 优化选项

优化选项的作用在于缩减代码规模和提高代码执行效率，常用的选项如下所述。

(1) -O、-O1：整个源代码会在编译、连接过程中进行优化处理，可执行文件的执行效率可以提高，但是编译、连接的速度会相应慢些。对于复杂函数，优化编译占用较多的时间和相当大的内存。在-O1 下，编译会尽量减少代码体积和代码运行时间，但是并不执行会花费大量时间的优化操作。

(2) -O2：除了不涉及空间和速度交换的优化选项，执行几乎所有优化工作。比-O 有更好的优化效果，但编译连接速度更慢。-O2 将会花费更多的编译时间，同时也会生成性能更好的代码，但并不执行循环展开和函数“内联”优化操作。

(3) -O3：在-O2 的基础上加入函数内联、循环展开和其他一些与处理器特性相关的优化工作。

下面通过 optimize.c 程序对比优化前后的效果。

```
#include <stdio.h>
int main(void)
{
    double counter;
    double result;
    double temp;
    for (counter = 0; counter < 2000 * 2000 * 2000 / 20.0 + 2020; counter += (5 - 1) / 4)
        {
            temp = counter / 1979;
            result = counter;
        }
        printf("Result is %lf\\n",result);
        return 0;
}
```

不加优化选项进行编译,程序执行耗时如下。

```
# gcc optimize.c -o optimize
# time ./optimize
Result is 400002019.000000\n
real    0m4.203s
user    0m4.190s
sys     0m0.020s
```

增加优化选项进行编译,程序执行耗时如下。

```
# gcc -O1 optimize.c -o optimize1
# time ./optimize1
Result is 400002019.000000\n
real    0m1.064s
user    0m1.060s
sys     0m0.010s
```

3.2　GDB 调试器

应用程序的调试是开发过程中必不可少的环节之一。Linux 操作系统下,GNU 的调试器称为 GDB(GUN Debugger),该软件最早由 Richard Stallman 编写,是一个用来调试 C 和 C++语言程序的调试器,它能使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 主要可以完成如下 4 个方面的功能。

(1) 启动程序,按照程序员自定义的要求运行程序。

(2) 单步执行、设置断点,可以让被调试的程序在所指定的断点处停住。

(3) 监视程序中变量的值。

(4) 动态地改变程序的执行环境。

3.2.1　GDB 基本使用方法

下面通过一个实例 test_g.c 介绍 GDB 的基本使用方法，test_g.c 文件的代码如下。

```
#include<stdio.h>
int sum(int n);
main()
{
    int s=0;
    int i,n;
    for(i=0;i<=50;i++)
    {
        s=i+s;
    }
    s=s+sum(20);
    printf("the result is %d\n",s);
}
int sum(int n)
{
    int total=0;
    int i;
    for(i=0;i<=n;i++)
        total=total+i;
    return (total);
}
```

使用 GDB 调试器，必须在编译时加入调试选项-g，命令如下。

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

生成可执行文件 test_g 后，启动 GDB 调试环境，命令如下。

```
#gdb test_g
GNU gdb (GDB) 7.5.91.20130417-cvs-ubuntu
Copyright (C) 2013 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 "x86_64-linux-gnu".
For bug reporting instructions,please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/test/test_g...done.
(gdb) l              //相当于 list,查看源代码
1    #include<stdio.h>
2    int sum(int n);
3    main()
4    {
5        int  s=0;
6        int i,n;
7        for(i=0;i<=50;i++)
```

```
8          {
9              s=i+s;
10         }
(gdb) l
11         s=s+sum(20);
12         printf("the result is   %d\n",s);
13     }
14     int sum(int n)
15     {
16         int total=0;
17         int i;
18         for(i=0;i<=n;i++)
19             total=total+i;
20     return (total);
(gdb) l
21     }
(gdb) break 7              //在源代码第 7 行设置断点
Breakpoint 1 at 0x400523: file test_g.c,line 7.
(gdb) break sum            //在源代码 sum 函数处设置断点
Breakpoint 2 at 0x400569: file test_g.c,line 16.
(gdb) info break           //显示断点信息
Num      Type              Disp Enb Address              What
1        breakpoint        keep y   0x0000000000400523 in main at test_g.c:7
2        breakpoint        keep y   0x0000000000400569 in sum at test_g.c:16
(gdb)  r                   //运行程序
Starting program: /home/test/test_g
Breakpoint 1,main () at test_g.c:7
7          for(i=0;i<=50;i++)
(gdb) n                    //在第一个断点处停止,n 相当于 next,单步执行
9                  s=i+s;
(gdb) n
7                  for(i=0;i<=50;i++)
(gdb) print s              //输出变量 s 的值
$1 = 0
(gdb) c                    //相当于 continue,继续执行
Continuing.
Breakpoin 2 ,sum(n=20) at test_g.c:16
16                 int total=0;
(gdb) c
Continuing.
the result is   1485
[Inferior 1 (process 2868) exited with code 024]
(gdb) q                    //退出 gdb
```

3.2.2　GDB 基本命令

GDB 命令很多,可以通过 help 命令来帮助查看了解,方法是启动 GDB 后输入 help 命令。

```
(gdb)help
List of classes of commands:
aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Command name abbreviations are allowed if unambiguous.
```

GDB 命令有很多，所以将它们分成了许多类。help 命令只列出了 GDB 的命令种类，如果要查看某一种类别下的具体命令，可以在 help 命令后加类名，具体格式如下。

```
help < class >
```

例如想了解 running 类下的具体命令，可以输入如下命令。

```
help running
```

常用的 GDB 命令如表 3.4 所示。

表 3.4　常用的 GDB 命令

命　令	描　述
backtrace	显示程序中的当前位置和表示如何到达当前位置的栈跟踪
break	设置断点
cd	改变当前工作目录
clear	清除停止处的断点
continue	从断点处开始继续执行
delete	删除一个断点或监视点
display	程序停止时显示变量或表达式
file	装入要调试的可执行文件
info	查看程序的各种信息
kill	终止正在调试的程序
list	列出源文件内容
make	使用户不退出 GDB 就可以重新产生可执行文件
next	执行一行代码，从而执行一个整体的函数
print	显示变量或表达式的值

续表

命　令	描　述
pwd	显示当前工作目录
quit	退出 GDB
run	执行当前被调试的程序
set	给变量赋值
shell	不离开 GDB 就执行 UNIX Shell 命令
step	执行一行代码并进入函数内部
watch	设置监视点，使用户能监视一个变量或表达式的值而不管它何时被变化

3.2.3　GDB 典型实例

例如如下程序，植入了错误，这里通过这个存在错误的程序介绍如何利用 GDB 进行程序调试。

bug.c 程序的功能是将输入的字符串逆序显示在屏幕上，源代码如下。

```
#include<stdio.h>
#include<string.h>
int main(void)
{
    int i,len;
    char str[]="hello";
    char *rev_string;
    len=strlen(str);
    rev_string=(char *)malloc(len+1);
    printf("%s\n",str);
    for(i=0;i<len;i++)
        rev_string[len-i]=str[i];
    rev_string[len+1]='\0';
    printf("the reverse string is%s\n",rev_string);
}
```

程序的编译和运行结果如下。

```
# gcc bug.c -o bug
# ./bug
hello
the reverse string is
```

以上运行的结果是错误的，正确结果如下。

```
hello
the reverse string is olleh
```

这时可以使用 GDB 调试器来查看问题出在哪儿，具体步骤是编译时加上-g 调试选项，然后再对可执行程序进行调试，主要通过单步执行，并同步查看 rev_string 字符数组中每个元素的值来寻找问题，具体命令如下。

```
# gcc bug.c -g -o bug
# gdb bug
```

执行命令后，进入调试环境，代码显示如下。

```
(gdb) l                          //列出源文件内容
1       #include<stdio.h>
2       #include<string.h>
3       int main(void)
4       {
5               int i,len;
6               char str[]="hello";
7               char *rev_string;
8               len=strlen(str);
9               rev_string=(char *)malloc(len+1);
10              printf("%s\n",str);
(gdb) l
11              for(i=0;i<len;i++)
12                      rev_string[len-i]=str[i];
13              rev_string[len+1]='\0';
14              printf("the reverse string is%s\n",rev_string);
15      }
(gdb) break 8                    //在源代码第 8 行设置断点
Breakpoint 1 at 0x4005f9: file bug.c,line 8.
(gdb) r                          //运行程序
Starting program: /home/test/bug
Breakpoint 1,main () at bug.c:8
8         len=strlen(str);
(gdb) n                          //在第一个断点处停止，n 相当于 next，单步执行
9               rev_string=(char *)malloc(len+1);
(gdb) print len                  //输出变量 len 的值
$1 = 5
(gdb) n                          //单步执行
10              printf("%s\n",str);
(gdb) n
hello
11              for(i=0;i<len;i++)
(gdb) n
12                      rev_string[len-i]=str[i];
(gdb) n
11              for(i=0;i<len;i++)
(gdb) n
12                      rev_string[len-i]=str[i];
(gdb) n
11              for(i=0;i<len;i++)
(gdb) print rev_string[5] //输出 rev_string[5]的值
$2 = 104 'h'
(gdb) print rev_string[4] //输出 rev_string[4]的值
$3 = 101 'e'
(gdb) n
```

```
12                          rev_string[len-i]=str[i];
(gdb) n
11              for(i=0;i<len;i++)
(gdb) print rev_string[3]
$4 = 108 'l'
(gdb) n
12                          rev_string[len-i]=str[i];
(gdb) n
11              for(i=0;i<len;i++)
(gdb) print rev_string[2]
$5 = 108 'l'
(gdb) n
12                          rev_string[len-i]=str[i];
(gdb) n
11              for(i=0;i<len;i++)
(gdb) print rev_string[1]
$6 = 111 'o'
(gdb) n
13              rev_string[len+1]='\0';
(gdb) n
14              printf("the reverse string is%s\n",rev_string);
(gdb) print rev_string[0]           //输出 rev_string[0]的值
$7 = 0 '\000'
(gdb) c                             //相当于 continue,继续执行
Continuing.
the reverse string is
```

通过以上调试过程可见,错误的根源在于没有给 rev_string[0]赋值,所以 rev_string[0]为'\0',导致字符串输出为空。将 rev_string[len-i]改成 rev_string[len-1-i],程序运行结果就是期待的结果。

3.3　Make 工具的使用

在大型软件项目的开发过程中,通常有成百上千个源文件,如 Linux 内核源文件。如果每次都通过手工键入 GCC 命令进行编译,非常不方便,Make 工具的引入解决了这个问题。Make 工具可以将大型的开发项目分解成为多个更易于管理的模块,简洁明快地理顺各个源文件之间纷繁复杂的相互依赖关系,最后自动完成编译工作。

Make 又叫工程管理器,即管理较多的工程文件。它最主要的功能是通过 Makefile 文件来描述源程序之间的相互依赖关系,并自动完成维护编译工作。Make 工具能够根据文件的时间戳自动发现更新过的文件,可以减少编译工作量。

Makefile 文件需要严格按照语法进行编写,文件中需要说明如何编译各个源文件并连接生成可执行文件,并定义源文件之间的依赖关系等。

3.3.1　Makefile

1. Makefile 基本结构

Makefile 可定义文件依赖关系，它由若干规则组成，格式如下。

```
target: dependency
<tab 键>　command
```

其中，target(目标体)是指 Make 工具最终需要创建的东西，通常是目标文件或可执行文件。另外，目标体也可以是一个 Make 工具执行的动作名称，如目标体 clean，可以称这样的目标体是“伪目标”。

dependency(依赖关系)是编译目标体要依赖的一个或多个文件列表。

command(命令)是指为了从指定的依赖关系中创建出目标体所需执行的命令。

一个规则可以有多个命令行，每一条命令占一行。注意，每一个命令的第一个字符必须是制表符 Tab，如果使用空格会导致错误，Make 工具会在执行过程中显示 Missing Separator(缺少分隔符)并停止。

本书 3.1 节中创建了一个名为 hello.c 的文件，并使用命令 gcc hello.c -o hello 生成了可执行文件 hello。如果要利用 Make 工具生成可执行程序，则首先要在 hello.c 所在的目录下编写一个 Makefile 文件，文件内容如下。

```
all: hello.o
    gcc  hello.o -o hello
hello.o:hello.c
    gcc -c hello.c -o hello.o
clean:
    rm *.o hello
```

上述 Makefile 文件共由 3 个规则组成，第一个规则的目标体名称是 all，依赖文件是 hello.o，命令的功能是生成 hello 文件；第二个规则的目标体名称是 hello.o，依赖文件是 hello.c，命令功能是生成 hello.o 目标文件；第三个规则的目标体名称是 clean，没有依赖文件，命令的功能是删除 *.o 和 hello 文件。

2. Make 工具的使用

Makefile 文件编写完成以后，需要通过 Make 工具来执行，使用 make 命令，格式如下。

```
make [target]
```

参数 target 是指要处理的目标体名，它是一个可选参数。make 命会自动查找当前目录下的 Makefile 或 makefile 文件，如果文件存在就执行，否则报错。如果 make 后面没有 target 参数，则执行 Makefile 文件的第一个目标体。

例如，如果使用上述编写好的 Makefile 文件，执行 make all 命令或 make 命令，都表示执行第一个目标体 all，即生成可执行文件 hello；执行 make clean 命令，表示执行第三个目标体 clean，即删除 hello 和扩展名为.o 的文件。

GUN Make 工具在当前工作目录中按照 GNUmakefile、makefile、Makefile 的顺序搜索 Makefile 文件，也可以通过-f 参数指定描述文件。如果编写的 Makefile 文件名为 zhs，则通

过 make -f zhs 命令即可执行。Make 工具的选项很多，读者可以到 Make 工具参考书中查阅。

3. Makefile 变量

嵌入式项目开发中经常使用交叉编译器，本书采用的交叉编译器是 arm-none-linux-gnueabi-gcc。如果要交叉编译 hello.c 程序，就要将 Makefile 文件中所有 gcc 替换成 arm-none-linux-gnueabi-gcc，如果一个一个修改会非常麻烦，所以可在 Makefile 中引进变量来解决。

为了简化编辑和维护 Makefile 文件，允许在文件中创建和使用变量，变量是指在 Makefile 文件中定义的名字，用来代替一个文本字符串，该文本字符串称为该变量的值。

Makefile 中的变量分为用户自定义变量、预定义变量和自动变量，用户自定义变量由用户自行设定，预定义变量和自动变量为在 Makefile 文件中经常使用的变量，其中部分有默认值，也就是常见的设定值，当然用户也可以对其进行修改。

使用变量将前文编写的 Makefile 文件改写成如下形式。

```
CC=gcc
OBJECT=hello.o
all:$(OBJECT)
    $(CC) $(OBJECT) -o hello
$(OBJECT):hello.c
    $(CC) -c hello.c -o $(OBJECT)
clean:
    rm *.o hello
```

在 Makefile 文件中，OBJECT 是用户自定义变量，它的值为 hello.o。CC 是预定义变量，它有默认值，但用户不想使用默认值，因此把 CC 的值修改为为 gcc。

变量的引用方法是把变量用括号括起来，并在前面加上 $。例如引用变量 CC，就可以写成 $(CC)。

变量一般都在 Makefile 文件的头部进行定义，按照惯例，变量名一般使用大写字母。变量的内容可以是命令、文件、目录、变量、文件列表、参数列表、常量或目标名等。

如果要对 3.1 节创建的 hello.c 进行交叉编译，可以将 Makefile 文件改写为如下形式。

```
CROSS= arm-none-linux-gnueabi-
CC= $(CROSS)gcc
OBJECT=hello.o
all:$(OBJECT)
    $(CC) $(OBJECT) -o hello
$(OBJECT):hello.c
    $(CC) -c hello.c -o $(OBJECT)
clean:
    rm *.o hello
```

预定义变量是 Make 工具预先定义好的变量，可以在 Makefile 文件中直接使用。引入预定义变量的目的是方便 Makefile 文件编写。预定义变量包括常见编译器、汇编器的名称及其编译选项等。常见预定义变量及其部分默认值如表 3.5 所示。

表 3.5　Makefile 中常见的预定义变量

变　量	含　义
AR	库文件维护程序的名称，默认值为 ar
AS	汇编程序的名称，默认值为 as
CC	C 编译器的名称，默认值为 cc
CPP	C 预编译器的名称，默认值为 $(CC) -E
CXX	C++编译器的名称，默认值为 g++
FC	FORTRAN 编译器的名称，默认值为 f77
RM	文件删除命令的名称，默认值为 rm -f
ARFLAGS	库文件维护程序的名称，无默认值
ASFLAGS	汇编程序的选项，无默认值
CFLAGS	C 编译器的选项，无默认值
CPPFLAGS	C 预编译的选项，无默认值
CXXFLAGS	C++编译器的选项，无默认值
FFLAGS	FORTRAN 编译器的选项，无默认值

自动变量是指可以表示编译语句中已出现的目标文件、依赖文件等信息的变量。引入自动变量的目的是进一步简化 Makefile 文件的编写。常见的自动变量如表 3.6 所示。

表 3.6　Makefile 中常见的自动变量

变　量	说　明
$@	规则的目标所对应的文件名称
$*	不包含扩展名的目标文件名称
$+	所有依赖文件，以空格分开，并以出现的先后为序，可能包含重复的依赖文件
$%	如果目标是归档成员，则该变量表示目标的归档成员名称
$<	规则中的第一个依赖文件名称
$^	规则中所有依赖的列表，以空格为分隔符
$?	规则中日期新于目标的所有依赖文件的列表，以空格为分隔符
$(@D)	目标文件的目录部分(如果目标在子目录中)
$(@F)	目标文件的文件名称部分(如果目标在子目录中)

3.3.2　Makefile 的应用

这里将通过实例来详细介绍 Makefile 的应用。

1. *所有文件均在一个目录下的 Makefile 文件*

例如现有 7 个文件，分别是 m.c、m.h、study.c、listen.c、visit.c、play.c 及 watch.c，各自内容如下。

m.c 文件内容如下。

```
#include<stdio.h>
main()
{
        int i;
        printf("please input the value of i from 1 to 5:\n");
```

```
        scanf("%d",&i);
        if(i==1)
            visit();
        else  if(i==2)
            study();
        else  if(i==3)
            play();
        else  if(i==4)
            watch();
        else  if(i==5)
            listen();
        else
            printf("nothing to do\n");
        printf("This is a woderful day\n");
}
```

study.c 文件内容如下。

```
#include<stdio.h>
void study()
{
        printf("study embedded system today\n");
}
```

listen.c 文件内容如下。

```
#include<stdio.h>
void listen()
{
    printf("listen english today\n");
}
```

play.c 文件内容如下。

```
#include<stdio.h>
void play()
{
    printf("play football today\n");
}
```

visit.c 文件内容如下。

```
#include<stdio.h>
void visit()
{
    printf("visit friend today\n");
}
```

watch.c 文件内容如下。

```
#include<stdio.h>
void watch()
{
```

```
    printf("watch TV today\n");
}
```

m. h 文件内容如下。

```
void visit();
void listen();
void watch();
void study();
void play();
```

从上述 7 个文件的代码可以看出它们之间的相互依赖关系，如图 3.2 所示。

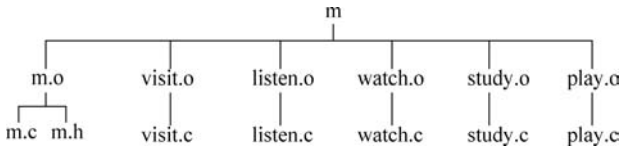

图 3.2　文件之间的依赖关系

现在利用这 7 个程序生成一个名为 m 的可执行程序，则 Makefile 文件可编写如下。

```
CC=gcc
TARGET=All
OBJECTS= m.o visit.o listen.o watch.o study.o play.o
$(TARGET):$(OBJECTS)
    $(CC)   $(OBJECTS) -o m
m.o:m.c m.h
    $(CC) -c m.c -o m.o
visit.o:visit.c
    $(CC) -c visit.c -o visit.o
listen.o:listen.c
    $(CC) -c listen.c -o listen.o
watch.o:watch.c
    $(CC) -c watch.c -o watch.o
study.o:study.c
    $(CC) -c study.c -o study.o
play.o:play.c
    $(CC) -c play.c -o play.o
clean:
    rm *.o
```

这个 Makefile 文件可以通过使用自动变量得以简化，现用 $@、$<、$^来改写上述 Makefile 文件如下。

```
CC=gcc
TARGET=All
OBJECTS= m.o visit.o listen.o watch.o study.o play.o
$(TARGET):$(OBJECTS)
    $(CC) $^-o m
m.o:m.c m.h
    $(CC) -c $< -o $@
```

```
visit.o:visit.c
    $(CC) -c $< -o $@
listen.o:listen.c
    $(CC) -c $< -o $@
watch.o:watch.c
    $(CC) -c $< -o $@
study.o:study.c
    $(CC) -c $< -o $@
play.o:play.c
    $(CC) -c $< -o $@
clean:
    rm *.o
```

从修改后的 Makefile 文件可以看出，各个文件的编译命令几乎没有区别，所以可进一步用%和 * 两个通配符来简化如下。

```
CC=gcc
TARGET=All
OBJECTS= m.o visit.o listen.o watch.o study.o play.o
$(TARGET):$(OBJECTS)
    $(CC) $^-o m
*.o:*.c
    $(CC) -c $< -o $@
clean:
    rm *.o
```

2. 文件在不同目录下的 Makefile 文件

假设程序的目录结构为源文件、可执行文件和 Makefile 在 src 目录中，头文件在 include 文件夹中，*.o 目标文件保存在 obj 目录中，如图 3.3 所示。

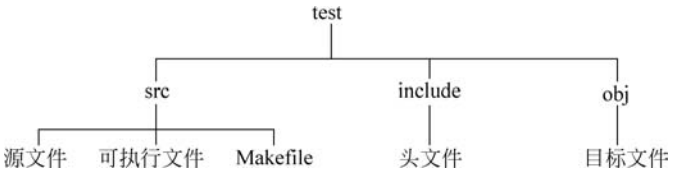

图 3.3　目录结构

程序文件分别保存在不同的目录下，所以在 Makefile 中需要指定目标文件和头文件的路径。仍以上述 7 个程序文件为例，Makefile 文件编写如下。

```
CC=gcc
SRC_DIR=./
OBJ_DIR=../obj/
INC_DIR=../include/
TARGET=all
$(TARGET):$(OBJ_DIR)m.o $(OBJ_DIR)visit.o $(OBJ_DIR)listen.o\
          $(OBJ_DIR)watch.o $(OBJ_DIR)study.o $(OBJ_DIR)play.o
          $(CC) $^-o $(SRC_DIR)m
$(OBJ_DIR)m.o:$(SRC_DIR)m.c  $(INC_DIR)m.h
          $(CC) -I$(INC_DIR) -c -o $@ $<
```

```
$(OBJ_DIR)visit.o: $(SRC_DIR)visit.c
        $(CC) -c $< -o $@
$(OBJ_DIR)listen.o: $(SRC_DIR)listen.c
        $(CC) -c $< -o $@
$(OBJ_DIR)watch.o: $(SRC_DIR)watch.c
        $(CC) -c $< -o $@
$(OBJ_DIR)study.o: $(SRC_DIR)study.c
        $(CC) -c $< -o $@
$(OBJ_DIR)play.o: $(SRC_DIR)play.c
        $(CC) -c $< -o $@
clean:
        rm $(OBJ_DIR)*.o
```

注意：在 Linux 操作系统下，../表示上一级目录。

3.3.3　自动生成 Makefile 文件

编写 Makefile 文件确实不是一件轻松的事，尤其对于一个较大的项目而言更是如此。autoTools 系列工具正是为编写 Makefile 文件而设的，它只需用户输入简单的目标文件、依赖文件、文件目录等就可以轻松地生成 Makefile 文件。另外，这些工具还可以完成系统配置信息的收集，用户可以方便地处理各种移植性的问题。

autoTools 包括 aclocal、autoscan、autoconf、autoheader 和 automake 工具等，使用 autoTools 主要就是利用各个工具的脚本文件来生成最后的 Makefile 文件，总体流程如图 3.4 所示。

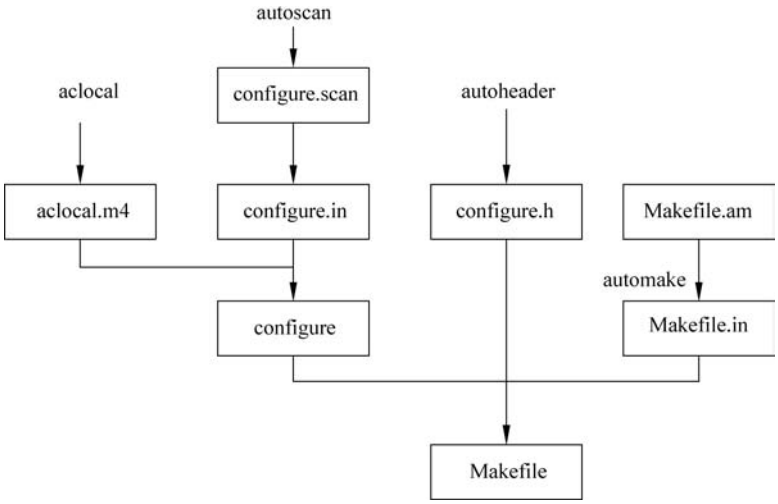

图 3.4　自动生成 Makefile 的流程图

以前文创建的 hello.c 为例，自动生成 Makefile 的过程如下所述。

1. autoscan

```
# ls
hello.c
```

```
# autoscan
# ls
autoscan.log   configure.scan   hello.c
```

2. 创建 configure.in 文件

configure.in 是 autoconf 的脚本配置文件，是在 configure.scan 的基础上修改而来的，代码如下。

```
# vi configure.scan
# -*- Autoconf -*-                          //以#开头的行为注释
AC_PREREQ(2.59)                             //本文件要求的 autoconf 版本
AC_INIT(hello,1.0)                          //AC_INIT 宏用来定义软件的名称和版本等信息
AM_INIT_AUTOMAKE(hello,1.0)                 //automake 所必备的宏、软件名称和版本号
AC_CONFIG_SRCDIR([hello.c])                 //用来侦测所指定的源码文件是否存在
AC_CONFIG_HEADER([config.h])                //用于生成 config.h 文件，以便 autoheader 使用
AC_PROG_CC
AC_CONFIG_FILES([Makefile])                 //用于生成相应的 Makefile 文件
AC_OUTPUT
```

最后用命令 mv configure.scan configure.in 将 configure.scan 改成 configure.in。

3. 运行 aclocal 生成 aclocal.m4 文件

```
# aclocal
# ls
aclocal.m4   autoscan.log   configure.in   hello.c
```

4. 运行 autoconf 生成 configure 可执行文件

```
# autoconf
# ls
aclocal.m4   autom4te.cache   autoscan.log   configure   configure.in   hello.c
```

5. 使用 autoheader 生成 config.h.in

```
# autoheader
```

6. 创建 Makefile.am 文件

automake 用的脚本配置文件是 Makefile.am，需要先创建相应的文件。

```
# vi Makefile.am
```

内容如下。

```
AUTOMAKE_OPTIONS=foreign
bin_PROGRAMS= hello
hello_SOURCES= hello.c
```

接下来用 automake 生成 Makefile.in，使用选项 adding-missing 可以让 automake 自动添加一些必需的脚本文件，命令如下。

```
# automake --add-missing
configure.in: installing './install-sh'
```

```
configure.in: installing './missing'
Makefile.am: installing 'depcomp
# ls
aclocal.m4   autoscan.log   configure   depcomp   install-sh   Makefile.in
autom4te.cache   config.h.in   configure.in   hello.c   Makefile.am   missing
```

7. configure

通过运行自动配置设置文件 configure，Makefile.in 即变成了最终的 Makefile 文件。

```
# ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking for gcc... gcc
checking for C compiler default output... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ANSI C... none needed
checking for style of include used by make... GNU
checking dependency style of gcc... gcc3
configure: creating ./config.status
config.status: creating Makefile
config.status: creating config.h
config.status: executing depfiles commands
```

8. 执行 make 命令生成可执行文件 hello

```
# make
cd . && /bin/sh ./config.status config.h
config.status: creating config.h
config.status: config.h is unchanged
make   all-am
make[1]: Entering directory '/lvli/12'
source='hello.c' object='hello.o' libtool=no \
depfile='.deps/hello.Po' tmpdepfile='.deps/hello.TPo' \
depmode=gcc3 /bin/sh ./depcomp \
gcc -DHAVE_CONFIG_H -I. -I. -I.       -g -O2 -c 'test -f 'hello.c' || echo './''hello.c
gcc   -g -O2    -o hello   hello.o
cd . && /bin/sh ./config.status config.h
config.status: creating config.h
config.status: config.h is unchanged
make[1]: Leaving directory '/lvli/12'
```

9. 运行 hello

```
# ./hello
hello,world!
```

3.4　Linux 应用程序设计

虽然 Linux 操作系统下的 C 语言编程与 Windows 操作系统下的 C 语言编程方法基本相同，但是也有细微的差别。这里通过文件操作、时间获取和多线程等任务介绍 Linux 应用程序设计。

3.4.1　文件操作编程

在 Linux 操作系统下，实现文件操作可以采用两种方法，一种是通过 C 语言库函数调用来实现，另一种是通过 Linux 系统调用来实现。前者独立于具体操作系统，即在任何操作系统下，使用 C 语言库函数操作文件的方法都相同，后者则依赖于 Linux 操作系统。

1. C 语言库函数

C 语言库提供了一系列用来操作文件的函数，这些函数的说明都包含在 stdio.h 头文件中。

1）打开和关闭文件函数

打开文件可通过 fopen 函数来完成，关闭文件可通过 fclose 函数来完成，格式如下。

```
FILE * fopen(const char * filename,const char * mode);
int fclose(FILE * stream);
```

其中，参数 filename 表示打开的文件名（包括路径，默认为当前路径）。mode 为文件打开模式，常见模式如表 3.7 所示。若成功打开文件，fopen 函数返回值是文件指针；若文件打开失败，则返回 NULL，并把错误代码存在 errno 中。

表 3.7　常见文件打开模式

模　　式	含　　义
r,rb	只读方式打开文件，该文件必须存在
r+,rb+	读写方式打开文件，若文件不存在则自动创建
w,wb	只写方式打开文件，若文件不存在则自动创建
w+,wb+	读写方式打开文件，若文件不存在则自动创建
a,ab	追加方式打开文件，若文件不存在则自动创建
a+,ab+	读和追加方式打开文件，若文件不存在则自动创建

模式名称中的 b 用于区分文本文件和二进制文件。在 Windows 操作系统下有区分，但在 Linux 下不需要区分。

2）读取文件数据函数

读取文件数据可通过 fread、fgetc 和 fgets 等函数实现，格式如下。

```
size_t fread(void * ptr,size_t size,size_t n,FILE * stream);
int fgetc(FILE * stream);
char * fgets(char * s,int size,FILE * stream);
```

fread 函数的功能是从 stream 指向的文件中读取长度为 $n \times$ size 字节的字符串，并将读

取的数据保存到 ptr 缓存中，返回值是实际读出数据的字节数。fgetc 函数的功能是从 stream 指向的文件中读取一个字符，若读到文件尾，则返回 EOF。fgets 函数的功能是从 stream 指向的文件中读取一串字符，并存到 s 缓存中，直到出现换行字符、文件尾或已读了 size－1 个字符时结束，最后会加上 NULL 作为字符串结束符。

3）向文件写数据函数

向文件写数据可通过 fwrite、fputc 和 fputs 等函数实现，格式如下。

```
size_t fwrite(const void * ptr, size_t size, size_t n, FILE * stream);
int fputc(int c, FILE * stream);
int fputs(const char * s, FILE * stream);
```

fwrite 函数的功能是将 ptr 缓存中的数据写到 stream 指向的文件中，写入长度为 $n \times$ size 字节，返回值是实际写入的字节数。fputc 函数的功能是向 stream 指向的文件中写入一个字符。fputs 函数的功能是将 s 缓存中的字符串写入 stream 指向的文件中。

【程序 3.1】 文件复制程序 file_copy.c。

```
#include <stdio.h>
#include <stdlib.h>
#define BUFFER_SIZE 1024
int main(int argc, char * * argv )
{
    FILE * fileFrom, * fileTo;
    char buffer[BUFFER_SIZE]={0};
    int length=0;
    /* 检查输入命令格式是否正确 */
    if(argc!=3)
        {    printf("Usage:%s fileFrom fileTo\n",argv[0]);
             exit(0);
        }
    /* 打开源文件 */
    fileFrom = fopen(argv[1],"rb+");
    if(fileFrom==NULL)
        {
            printf(" Open File %s Failed\n",argv[1]);
            exit(0);
        }
    /* 打开或创建目标文件 */
    fileTo = fopen(argv[2],"wb+");
    if(fileTo==NULL)
        {
            printf(" Open File %s Failed\n",argv[2]);
            exit(0);
            }
    /* 复制文件内容 */
    while ((length =fread(buffer,1,BUFFER_SIZE,fileFrom))>0 )
        {
              fwrite(buffer,1,length,fileTo);
        }
```

```
    /*关闭文件*/
    fclose(fileFrom);
    fclose(fileTo);
    return 0;
}
```

编译源程序 file_copy.c，生成可执行程序 file_copy，然后执行 file_copy 程序将 hello.c 复制成 zhs.c，编译和运行命令如下。

```
#gcc  file_copy.c  -o  file_copy
#./file_copy  hello.c  zhs.c
```

2. 利用 Linux 系统调用函数完成文件操作

C 语言库中的 fopen、fclose、fwrite、fread 等函数其实是由操作系统的 API 函数封装而来的，如 fopen 内部其实调用的是 open 函数，fwrite 内部调用的是 write 函数。用户也可以直接利用 Linux 操作系统的 API 函数来完成文件操作编程，常用的有 open、close、write、read 等。

【程序 3.2】 文件创建程序 file_create.c。

```
#include<stdio.h>
#include<string.h>
#include<fcntl.h>
#define MAX 40
main()
{
    int fd,n,ret;
    char writebuf[MAX]="This is a test data!";
    /*打开文件,如果文件不存在,则创建文件*/
    fd = open("a.txt",O_RDWR | O_CREAT);
      if (fd<0)
      {
          perror("Open File Error!");
          return 1;
      }
    /*向文件写入字符串*/
    ret = write(fd,writebuf,strlen(writebuf));
     if (ret<0)
      {
          perror("Write Error!");
          return 1;
     }
     else
     {
         printf("write %d characters!\n",ret);
     }
    /*关闭时,会自动保存文件*/
    close(fd);
}
```

编译源程序 file_create.c，生成可执行程序 file_create，然后执行如下命令，执行完成后

就会创建一个名为 a.txt 的文本文件，文件内容为 This is a test data!。

```
#gcc file_create.c -o file_create
#./file_create
write 20 characters!
```

3.4.2　时间编程

在编程中经常要使用到时间，如获取系统时间、计算事件耗时等。这时需要用到时间函数，这些函数的说明包含在 time.h 头文件中。

1. time 函数

函数格式：

```
time_t time(time_t *tloc);
```

函数功能：获取日历时间，即从 1970 年 1 月 1 日 0 点到现在所经历的秒数，结果保存在 tloc 中，如果操作成功，则返回值为经历的秒数；若操作失败，则返回值为((time_t)－1)，错误原因存于 errno 中。

2. gmtime 函数

函数格式：

```
struct tm *gmtime(const time_t *timep);
```

函数功能：将日历时间转化为格林威治标准时间，并将数据保存在 tm 结构中。tm 结构的定义如下。

```
struct tm
{
int tm_sec;                //秒
int tm_min;                //分
int tm_hour;               //时
int tm_mday;               //日
int tm_mon;                //月
int tm_year;               //年
int tm_wday;               //本周第几日
int tm_yday;               //本月第几日
int tm_isdst;              //日光节约时间
};
```

3. gettimeofday 函数

函数格式：

```
int gettimeofday(struct timeval *tv,struct timezone *tz);
```

函数功能：获取从今日凌晨到现在的时间差，并存放在 tv 结构中，然后将当地时区的信息存放到 tz 结构中。tv 和 tz 两个结构的定义如下。

```
strut timeval {
long tv_sec;               //秒数
long tv_usec;              //微秒数
```

```
};
struct timezone{
int tz_minuteswest;        //和 GMT 的时间差
int tz_dsttime;
};
```

4. sleep 和 usleep 函数

函数格式：

```
unsigned int sleep(unsigned int sec);
void usleep(unsigned long usec);
```

函数功能：sleep 函数的功能是使程序睡眠 sec 秒，usleep 函数的功能是使程序睡眠 usec 微秒。

【程序 3.3】 算法分析程序 test_time.c

```
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* 算法 */
void function()
{
    unsigned int i,j;
    double y;
    for(i=0;i<100;i++)
        for(j=0;j<100;j++)
            {usleep(10);y++;}
}

main()
{
    struct timeval tpstart,tpend;
    float timeuse;

    gettimeofday(&tpstart,NULL);        //开始时间
    function();
    gettimeofday(&tpend,NULL);          //结束时间
    /* 计算算法执行时间 */
    timeuse=1000000*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec;
    timeuse/=1000000;
    printf("Used Time:%f sec.\n",timeuse);
    exit(0);
}
```

程序编译及运行结果如下。

```
#gcc  test_time.c  -o  test_time
#./test_time
Used Time : 39.432861 sec.
```

3.4.3　多线程编程

1. 线程编程基础

进程是系统分配资源的最小单位，线程是系统调度的最小单位。线程是进程中的某一个能独立运行的程序片段。在 Linux 系统下，启动一个新进程必须分配给它独立的地址空间，建立众多的数据表来维护它的代码段、堆栈段和数据段，这是一种“昂贵”的多任务工作方式。而启动一个新线程则不需要这些操作，所以线程是一个非常“节俭”的多任务操作方式。另外，线程之间的通信非常方便，因为线程是在同一进程中，一个线程可以直接访问另一个线程的数据。目前，实际应用中比较普遍采用多线程编程，因为可以提高程序的运行效率。

目前，绝大多数嵌入式操作系统和中间件都支持多线程。Linux 操作系统的多线程遵循 POSIX 线程接口，称为 pthread。在 Linux 操作系统下进行多线程编程时，需要使用 pthread.h 头文件以及 libpthread.so 和 libpthread.a 库文件。库文件中有许多与线程相关的文件，下面介绍 3 个常用的线程函数。

1）pthread_create 函数

函数格式：

```
int pthread_create(pthread_t * tid,const pthread_attr_t * attr,void * ( * start_rtn) (void ),void * arg)
```

函数功能：创建一个新的线程。参数 tid 为线程 id；attr 为线程属性，通常设置为 NULL；start_rtn 是线程要执行的函数；arg 是执行函数 start_rtn 的参数。当创建线程成功后，函数返回值为 0；若返回值为 EAGAIN，则表示系统限制创建新的线程，例如线程数目过多了；若返回值为 EINVAL，则表示第二个参数代表的线程属性值非法。创建线程成功后，新创建的线程则运行第三个参数和第四个参数确定的函数，原来的线程则继续运行下一行代码。

2）pthread_exit 函数

函数格式：

```
int pthread_exit(void * rval_ptr)
```

函数功能：退出当前线程，返回值保存在 rval_ptr 中。

3）pthread_join 函数

函数格式：

```
int pthread_join(pthread_t tid,void ** rval_ptr);
```

函数功能：阻塞调用线程，直到指定的线程终止。参数 tid 是指定的线程，rval_ptr 是线程退出的返回值。

【程序 3.4】 创建一个线程程序，程序文件名为 p-1.c。

```
#include < pthread.h >
#include < unistd.h >
#include < stdio.h >
/ * 子线程执行的函数 * /
```

```
void * thread(void * str)
{
    int i;
    for (i = 0; i < 6; ++i)
    {
        sleep(2);
        printf( "This in the thread: %d\n" ,i );
    }
    return NULL;
}

int main()
{
    pthread_t pth;
    int i;
    int ret;
    /* 创建一个子线程 pth,子线程执行 thread 函数中的程序 */
    ret=pthread_create(&pth,NULL,thread,(void * )(i));
    if(ret)
    {
        printf("Create pthread error!\n");
        return 1;
    }
    printf("Test start\n");
    for (i = 0; i < 6; ++i)
    {
        sleep(1);
        printf( "This in the main: %d\n" ,i );
    }

   pthread_join(pth,NULL);                    //等待线程结束
   return 0;
}
```

程序编译及运行结果如下：

```
# gcc p-1.c -lpthread -o p-1
# ./p-1
Test start
This in the main: 0      //主线程上的输出
This in the thread: 0    //子线程上的输出
This in the main: 1      //主线程上的输出
This in the main: 2      //主线程上的输出
This in the thread: 1    //子线程上的输出
This in the main: 3      //主线程上的输出
This in the main: 4      //主线程上的输出
This in the thread: 2    //子线程上的输出
This in the main: 5      //主线程上的输出
This in the thread: 3    //子线程上的输出
This in the thread: 4    //子线程上的输出
This in the thread: 5    //子线程上的输出
```

2. 互斥锁编程

因为多线程经常需要共享进程中的资源和地址空间，因此在对这些资源进行操作时，必须考虑到线程间资源访问的同步与互斥问题。在 POSIX 中有两种线程同步机制，分别为互斥锁和信号量，这两个同步机制可以互相通过调用对方来实现，但互斥锁更适用于同时可用的资源唯一的情况，而信号量更适用于同时可用的资源为多个的情况。

互斥锁是用一种简单的加锁方法来控制对共享资源的原子操作。这个互斥锁只有两种状态，也就是上锁和解锁。有时可以把互斥锁看作某种意义上的全局变量，在同一时刻只能有一个线程掌握互斥锁，拥有上锁状态的线程能够对共享资源进行操作。若其他线程希望上锁一个已经被上锁的互斥锁，则该线程就会挂起，直到上锁的线程释放互斥锁为止。因此，互斥锁可以保证让每个线程对共享资源按顺序进行原子操作。

1）pthread_mutex_init 函数

函数格式：

```
int pthread_mutex_init(pthread_mutex_t * mutex,const pthread_mutexattr_t * mutexattr)
```

函数功能：互斥锁初始化。参数 mutex 是互斥锁（又称互斥变量），mutexattr 是互斥锁的属性。如果参数 mutexattr 为 NULL，则使用默认的互斥锁属性，默认属性为快速互斥锁。当互斥锁初始化成功后，函数返回值为 0，否则其他任何返回值都表示出现了错误。

2）pthread_mutex_lock 函数

函数格式：

```
int pthread_mutex_lock(pthread_mutex_t * mutex)
```

函数功能：互斥锁上锁，如果该互斥锁已被另一个线程锁定和拥有，则调用该线程时将阻塞，直到该互斥锁变为可用为止。当互斥锁上锁成功后，函数返回值为 0，否则其他任何返回值都表示出现了错误。

3）pthread_mutex_unlock 函数

函数格式：

```
int pthread_mutex_unlock(pthread_mutex_t * mutex)
```

函数功能：互斥锁解锁（或称释放）。当互斥锁解锁成功后，函数返回值为 0，否则其他任何返回值都表示出现了错误。

【程序 3.5】 多个线程共享资源程序（不加互斥锁），程序名为 p-2.c。

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
 /* 多线程共享的全局变量 */
int sharei = 0;
void increase_num(void);

int main()
{
   int ret;
```

```
    pthread_t thread1,thread2,thread3;
    ret = pthread_create(&thread1,NULL,(void *)&increase_num,NULL);
    ret = pthread_create(&thread2,NULL,(void *)&increase_num,NULL);
    ret = pthread_create(&thread3,NULL,(void *)&increase_num,NULL);

    pthread_join(thread1,NULL);
    pthread_join(thread2,NULL);
    pthread_join(thread3,NULL);

    printf("sharei = %d\n",sharei);

    return 0;
}

void increase_num(void)
{
    long i,tmp;
    for(i =0;i<10000;i++)
    {
        tmp = sharei;
        tmp = tmp + 1;
        sharei = tmp;
    }
}
```

程序编译及运行结果如下。

```
# gcc p-2.c -lpthread -o p-2
# ./p-2
sharei = 30000      //第 1 次运行结果
# ./p-2
sharei = 24514      //第 2 次运行结果
# ./p-2
sharei = 23688      //第 3 次运行结果
```

该程序有 3 个线程，全局变量 sharei 是 3 个线程的共享资源，每个线程都通过调用 increase_num 函数来修改全局变量 sharei 的值，每个线程都将 sharei 的值增加 10000，而全局变量 sharei 的初始值为 0，所以程序运行结束后，全局变量 sharei 的值应该为 30000。但该程序实际运行时，每次运行结果都不一样。产生这种现象的原因是没有对全局变量赋值过程进行锁定，导致程序运行结果不确定。

【程序 3.6】 多个线程共享资源程序(带互斥锁)，程序名为 p-3.c。

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
 /* 多线程共享全局变量 */
int sharei = 0;
void increase_num(void);
/* 互斥锁 */
```

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int main()
{
  int ret;
  pthread_t thread1,thread2,thread3;
  ret = pthread_create(&thread1,NULL,(void *)&increase_num,NULL);
  ret = pthread_create(&thread2,NULL,(void *)&increase_num,NULL);
  ret = pthread_create(&thread3,NULL,(void *)&increase_num,NULL);

  pthread_join(thread1,NULL);
  pthread_join(thread2,NULL);
  pthread_join(thread3,NULL);

  printf("sharei = %d\n",sharei);

  return 0;
}
void increase_num(void)
{
  long i,tmp;
  for(i =0;i<10000;i++)
  {
    /*加锁*/
    if(pthread_mutex_lock(&mutex) != 0)
    {
      perror("pthread_mutex_lock");
      exit(EXIT_FAILURE);
    }
    tmp = sharei;
    tmp = tmp + 1;
    sharei = tmp;
    /*解锁*/
    if(pthread_mutex_unlock(&mutex) != 0)
    {
      perror("pthread_mutex_unlock");
      exit(EXIT_FAILURE);
    }
  }
}
```

程序编译及运行结果如下。

```
# gcc p-3.c -lpthread -o p-3
# ./p-3
sharei = 30000      //第 1 次运行结果
# ./p-3
sharei = 30000      //第 2 次运行结果
```

添加互斥锁后,多次运行的结果都是一样的。

3.5 练 习 题

1. 选择题

(1) GCC 软件是(　　)。

A. 调试器　　B. 编译器　　C. 文本编辑器　　D. 连接器

(2) GCC 支持的文件类型比较多,但不包括(　　)。

A. .c　　B. .o　　C. .h　　D. .t

(3) 在 Linux 系统中,./表示 (　　)。

A. 当前目标　　B. 上一级目录　　C. 根目录　　D. 用户目录

(4) 在 Linux 系统中,../表示(　　)。

A. 当前目标　　B. 上一级目录　　C. 根目录　　D. 用户目录

(5) GCC 编译 4 个过程中,汇编的主要功能是(　　)。

A. 将文件转换成汇编语言　　B. 将汇编语言代码转换成目标代码

C. 将汇编语言代码转换成可执行程序　　D. 连接目标代码转换成可执程序

(6) Linux 操作系统下的库文件都是以(　　)字母开头。

A. inc　　B. lin　　C. src　　D. lib

(7) GCC 用于指定头文件目录的选项是(　　)。

A. -o　　B. -L　　C. -g　　D. -I

(8) 若要用 GDB 调试,则用 GCC 编译时要加入调试选项(　　)。

A. -o　　B. -L　　C. -g　　D. -I

(9) GDB 软件是(　　)。

A. 调试器　　B. 编译器　　C. 文本编辑器　　D. 连接器

(10) 若要生成计算机上(Linux 操作系统)能够执行的程序,则使用的 C 编译是(　　)。

A. TC　　B. VC

C. GCC　　D. arm-none-linux-gnueabi-gcc

(11) Make 工具能够根据(　　)自动发现更新过的文件,从而减少编译工作量。

A. 文件的时间戳　　B. 文件创建时间　　C. 系统时间　　D. 当前时间

(12) Make 工具也可以通过(　　)参数指定描述文件。

A. -f　　B. -g　　C. -l　　D. -o

(13) Makefile 中的变量分为 3 类,(　　)不属于其中。

A. 用户自定义变量　B. 系统定义　　C. 预定义变量　　D. 自动变量

(14) CC 是(　　)。

A. 用户自定义变量　B. 系统定义　　C. 预定义变量　　D. 自动变量

(15) 变量的引用方法是把变量用括号括起来,并在前面加上(　　)。

A. $　　B. #　　C. *　　D. “

(16) Makefile 有许多自动变量,表示目标名称的是(　　)。

A. $@　　B. $^　　C. $<　　D. $>

(17) Makefile 有许多自动变量,表示第一个依赖文件的是(　　)。

A. $@　　B. $^　　C. $<　　D. $>

(18) Makefile 有许多自动变量,表示所有依赖文件的是(　　)。

A. $@　　B. $^　　C. $<　　D. $>

(19) sleep 函数功能是(　　)。

A. 唤醒程序　　B. 获取时间　　C. 创建线程　　D. 程序睡眠

(20) 在 Linux 操作系统下进行多线程编程要使用的库文件是(　　)。

A. libpthread.h　　B. pthread_join　　C. libpthread.a　　D. pthread_exit

2. 填空题

(1) GCC 编译 C 语言生成可执行文件要经历_____、_____、_____和_____4 个过程。

(2) Linux 操作系统下,动态链接库文件以_____结尾,静态链接库文件以_____结尾。动态链接库是在_____动态加载的,静态链接库是在_____静态加载的。

(3) GDB 是一个用来调试_____和_____语言程序的调试器。

(4) GDB 中,列出源文件内容的命令是_____,设置断点的命令是_____,运行程序的命令是_____,单步执行的命令是_____。

(5) 在编辑 Makefile 时,引用变量只需在变量前面加上_____符号。

(6) Makefile 中的变量分为 3 类,即_____、_____和_____。

(7) Makefile 文件中 OBJ 是自定义变量,$@是_____变量,CFLAGS 是_____变量。

(8) Makefile 文件中的预定义变量 CC 表示_____,CPP 表示_____,AR 表示_____,AS 表示_____。

3. 简答题

(1) 简述 GCC 编译 4 个过程中预编译的主要功能。

(2) 简述 GCC 编译 4 个过程中编译的主要功能。

(3) 简述 GCC 编译 4 个过程中连接的主要功能。

(4) 简述 GDB 主要完成的功能。

(5) 简述 Makefile 的基本结构。

(6) 简述用 C 语言实现文件操作可以采用那两种方法。

(7) 简述 fwrite、fputc 和 fputs 函数的功能。

(8) 简述 gmtime 函数和 gettimeofday 函数的功能。

(9) 简述 pthread_create 函数和 pthread_join 函数的功能。

(10) 简述进程和线程之间的区别。

(11) 简述在 POSIX 中有哪几种线程同步机制,它们之间有什么区别?

4. 编程及调试题

(1) 根据要求编写 Makefile 文件。有 5 个文件分别是 main.c、visit.h、study.h、visit.c、study.c,具体代码如下。

main.c 文件：

```
#include<stdio.h>
main()
{
        int i;
        printf("please input the value of i from 1 to    5:\n");
        scanf("%d",&i);
        if(i==1)
                visit();
       if(i==2)
                study();
}
```

visit.h 文件：

```
void visit();
```

study.h 文件：

```
void study();
```

visit.c 文件：

```
#include "visit.h"
void visit()
{
    printf("visit friend today\n");
}
```

study.c 文件：

```
#include "study.h"
void study()
{
        printf("study embedded system today\n");
}
```

① 如果上述 5 个文件在同一目录下，请编写 Makefile 文件。

② 如果按照下面的目录结构存放文件，请编写 Makefile 文件。

---bin：存放生成的可执行文件。

---obj：存放.o 文件。

---include：存放 visit.h、study.h。

---src：存放 main.c、visit.c、study.c 和 Makefile。

③ 如果按照下面的目录结构存放文件，请编写 Makefile 文件。

---bin：存放生成的可执行文件。

---obj：存放.o 文件。

---include：存放 visit.h、study.h。

---src：存放 main.c 和 Makefile。

---src1：存放 visit.c。

---src2：存放 study.c。

(2) 按照要求完成以下操作。

① 用 vi 编辑一文件 test.c，其内容如下。

```
#include<stdio.h>
int main()
{
    int s=0,i;
    for(i=1;i<=15;i++)
    {
        s=s+i;
        printf("the value of s is %d \n",s);
    }
    return 0;
}
```

② 使用 gcc test.c -o test.o 编译，生成 test.o。

③ 使用 gcc test.c -g -o test1.o 编译，生成 test1.o。

④ 比较 test.o 和 test1.o 文件的大小，思考为什么？

(3) 使用 GDB 调试(2)题中的程序。

① 带调试参数-g 进行编译。

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

② 启动 GDB 调试，开始调试。

```
#gdb  test
```

③ 使用 GDB 命令进行调试。

(4) 编写一个程序，将系统时间以“year-month-day hour：minute：second”格式保存在 time.txt 文件中。