武汉纺织大学
WUHAN TEXTILE UNIVERSITY

崇真尚美

Linux系统编程

数学与计算机学院

教师：朱萍

本课程的主要内容

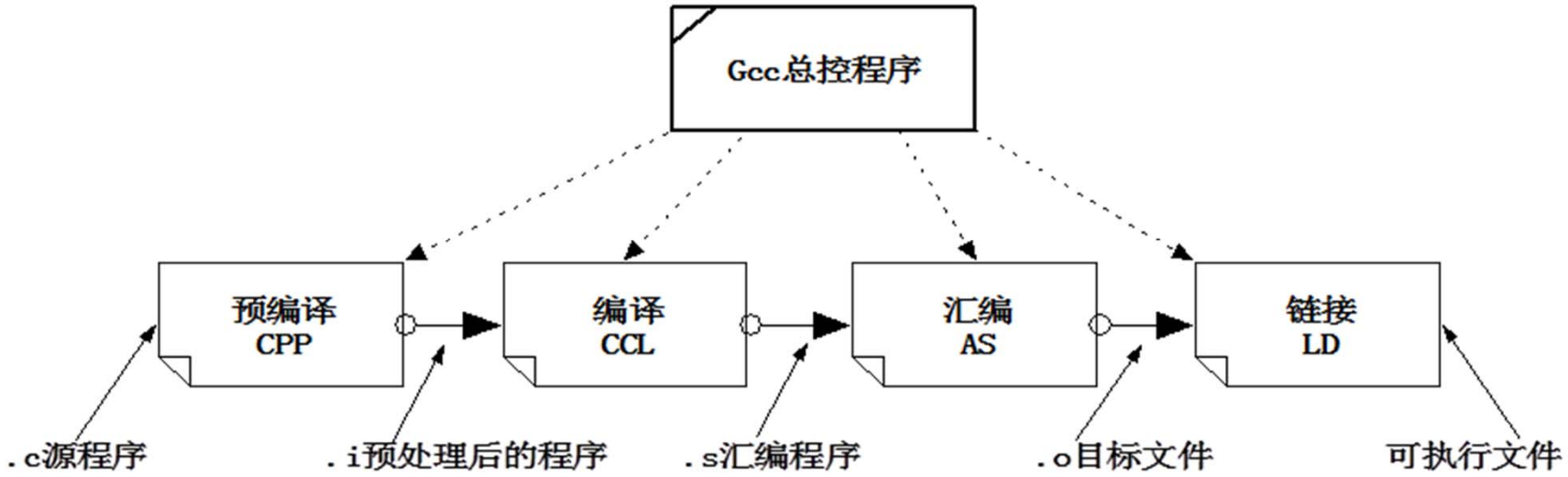
- Linux基础知识（4学时）
- Linux文件系统（6学时）
- Linux系统管理（2学时）
- Linux网络管理及应用（2学时）
- Linux Shell编程（6学时）
- Linux下C编程（6学时）
- Linux下进程通信（6学时）
- Linux下线程通信（4学时）
- Linux文件接口编程（6学时）
- Linux图形系统概述（2学时）
- Linux 网络编程（4学时）

Linux C编程

- GCC简介
- GCC的使用
- GCC编译举例
- 程序调试：gdb
- 程序维护：Makefile文件及make

GCC简介

- gcc（GNU Compliler Collection），是GNU推出的功能强大、性能优越的多平台编译器，是GNU的代表作品之一。
- gcc编译器能将C、Fortran、 C++、java语言源程序、汇编程序编译、连接成可执行文件。
- gcc还能支持各种不同的目标体系结构。常见的有X86系列、ARM、PowerPC等。
- gcc还能运行在不同的操作系统上，如Linux、Solaris、Windows等。


Gcc总控程序
预编译
CPP
编译
CCL
汇编
AS
链接
LD
.c源程序
.i预处理后的程序
.s汇编程序
.o目标文件
可执行文件

Linux C编程

- GCC简介
- GCC的使用
- GCC编译举例
- 程序调试：gdb
- 程序维护：Makefile文件及make

GCC的使用

（1）gcc –o helloworld helloworld.c

将helloworld.c编译成可执行文件helloworld。

（2）gcc testfun.c test.c -o test

将文件testfun.c 和test.c 编译成可执行test。

GCC语法格式 ：

gcc [options] [filenames]

options：编译器所需要的编译选项

filenames：要编译的文件名

超过100个！！！
不必每个记住，记住常用的就行

GCC的编译常用选项

- ✓ -c：只编译生成目标文件(.o)，不链接成为可执行文件。
- ✓ -o output_filename：确定输出文件的名称。如果不给出这个选项，gcc就给出默认的可执行文件a.out。
- ✓ -S：生成汇编文件（注意，S大写）
- ✓ -g：产生符号调试工具(GNU的gdb)所必要的调试信息，要使用gdb对源代码进行调试，就必须加入这个选项。
- ✓ -O：对程序进行优化编译、连接，采用这个选项，整个源代码会在编译、连接过程中进行优化处理，这样产生的可执行文件的执行效率可以提高，但是，编译、连接的速度就相应地要慢一些。
- ✓ -O2：比-O更好的优化编译、连接，当然整个编译、连接过程会更慢。

GCC的编译常用选项

- ✓ -w：禁止所有警告信息
- ✓ -Wall： 允许发出GCC提供的所有有用警告信息
- ✓ -Werror ：视警告为错误，出现任何警告即放弃编译

bad. c

```
#include <stdio.h>
int main (void)
{
  printf ("Two plus two is %f\n", 4);
  return 0;
}
```

Gcc编译选项

```
jkx@ubuntu:~/work/chapt6$ gcc -o bad bad.c
bad.c: In function ‘main’:
bad.c:4:12: warning: format ‘%f’ expects argument of type ‘doub
 2 has type ‘int’ [-Wformat=]
    printf ("Two plus two is %f\n", 4);
            ^
jkx@ubuntu:~/work/chapt6$ ls
bad  bad.c
```

```
jkx@ubuntu:~/work/chapt6$ rm bad
jkx@ubuntu:~/work/chapt6$ gcc -o bad bad.c -w
jkx@ubuntu:~/work/chapt6$ ls
bad  bad.c
```

Linux C编程

- GCC简介
- GCC的使用
- GCC编译举例
- 程序调试：gdb
- 程序维护：Makefile文件及make

GCC 编译举例

main.c

```
#include "myinc.h“
int main(void)
{
   int i;
   i = fun1(1,2);
   fun2(i);
   return 0;
}
```

fun1.c

```
#include "myinc.h“
int fun1(int x,inty)
{
   return(x+y);
}
```

myinc.h

```
#include <stdio.h>
int fun1(int x,int y);
void fun2(int x);
```

fun2.c

```
#include <stdio.h>
#include "myinc.h“
void fun2(int x)
{
   printf("The result is%d\n",x);
}
```

```
jkx@ubuntu:~/work/chapt6/myfun$ gcc -o main main.c fun1.
jkx@ubuntu:~/work/chapt6/myfun$ ./main
the result is 3
```

C 头文件的编译选项

1、#include <stdio.h>

2、#include “myinc.h”

其中，第一类使用尖括号< >，第二类使用双引号“ ”

对于第一类类，gcc会在系统预设包含文件目录(如/usr/include)中搜寻相应的文件。

对于第二类，gcc首先当前目录中搜寻头文件，如果在当前目录中没有找到需要的文件，就需要在编译选项中使用-Idirname，到指定的dirname目录中去寻找头文件。

❖如果把myinc.h移到别的地方

```
jkx@ubuntu:~/work/chapt6/myfun$ ls
fun1.c  fun2.c  main.c  myinc.h
jkx@ubuntu:~/work/chapt6/myfun$ mv myinc.h ../
jkx@ubuntu:~/work/chapt6/myfun$ gcc -o main main.c fun1.c fun2.c
main.c:1:19: fatal error: myinc.h: No such file or directory
compilation terminated.
fun1.c:1:19: fatal error: myinc.h: No such file or directory
compilation terminated.
fun2.c:1:19: fatal error: myinc.h: No such file or directory
compilation terminated.
```

❖此时可以使用-Idirname

```
jkx@ubuntu:~/work/chapt6/myfun$ gcc -o main main.c fun1.c fun2.c -I..
jkx@ubuntu:~/work/chapt6/myfun$
```

静态链接库

- 把fun1.c和fun2.c的功能做成一个库文件，gcc的时候就不需要把fun1.c 和fun2.c再编译了，只需要链接这个库文件

```
jkx@ubuntu:~/work/chapt6/myfun$ ls
fun1.c  fun2.c  main.c  myinc.h
jkx@ubuntu:~/work/chapt6/myfun$ gcc -c fun1.c
jkx@ubuntu:~/work/chapt6/myfun$ gcc -c fun2.c
jkx@ubuntu:~/work/chapt6/myfun$ ls
fun1.c  fun1.o  fun2.c  fun2.o  main.c  myinc.h
jkx@ubuntu:~/work/chapt6/myfun$ ar -rc libmyfun.a fun1.o fun2.o
jkx@ubuntu:~/work/chapt6/myfun$ ls
fun1.c  fun1.o  fun2.c  fun2.o  libmyfun.a  main.c  myinc.h
jkx@ubuntu:~/work/chapt6/myfun$ gcc -o main main.c -L./ -lmyfun
jkx@ubuntu:~/work/chapt6/myfun$ ls
fun1.c  fun1.o  fun2.c  fun2.o  libmyfun.a  main  main.c  myinc.h
jkx@ubuntu:~/work/chapt6/myfun$ ./main
the result is 3
```

制作静态链接库步骤

步骤1：把原始的.c文件编译成.o文件

```
jkx@ubuntu:~/work/chapt6/myfun$ gcc -c fun1.c
jkx@ubuntu:~/work/chapt6/myfun$ gcc -c fun2.c
jkx@ubuntu:~/work/chapt6/myfun$ ls
fun1.c  fun1.o  fun2.c  fun2.o  main.c  myinc.h
```

步骤2：用ar命令归档，生成文件libmyfun.a（归档文件名一定要以lib打头, .a结尾）。语法格式：ar -rc <生成的档案文件名> <.o文件名列表>。

```
jkx@ubuntu:~/work/chapt6/myfun$ ar -rc libmyfun.a fun1.o fun2
jkx@ubuntu:~/work/chapt6/myfun$ ls
fun1.c  fun1.o  fun2.c  fun2.o  libmyfun.a  main.c  myinc.h
```

使用静态链接库的方法

- 把静态库文件的路径加到**-L**参数里面，把库文件名（去掉打头的**lib**和结尾的**.a)**加到**-l**参数后面。

```
jkx@ubuntu:~/work/chapt6/myfun$ gcc -o main main.c -L./
jkx@ubuntu:~/work/chapt6/myfun$ ls
fun1.c  fun1.o  fun2.c  fun2.o  libmyfun.a  main  main.c
```

小写l

动态链接库

- 静态库和动态库的区别：
  - 有点类似动态编译和静态编译
  - 静态库最后生成的可执行文件中包含了静态库，在运行的目标机上不需要再存储这个静态库了
  - 动态库生成的可执行文件不包含库文件，只是留好了记号，运行时去寻找什么库文件，要求目标机中有此动态库文件
  - 一般来说，静态库编译出来的可执行文件大小要大于动态库编译出来的可执行文件。

制作动态链接库

- 直接用gcc命令的-shared –fPIC选项

```
/chapt6/myfun$ gcc -o libmyfun.so -fPIC -shared fun1.c fun2.
/chapt6/myfun$ ls
ibmyfun.so  main.c  myinc.h
```

使用动态链接库

```
jkx@ubuntu:~/work/chapt6/myfun$ gcc -o main main.c -L./ -lmyfun
jkx@ubuntu:~/work/chapt6/myfun$ ./main
./main: error while loading shared libraries: libmyfun.so: cannot open
ject file: No such file or directory
jkx@ubuntu:~/work/chapt6/myfun$ export LD_LIBRARY_PATH=./:$LD_LIBRARY_
jkx@ubuntu:~/work/chapt6/myfun$ ./main
the result is 3
```

把当前目录加入到这个环境变量中，**:$LD_LIBRARY_PAT**表示原有的值保持不变

- 运行可执行文件，需要设置环境变量 **LD_LIBRARY_PATH**告诉系统动态库在哪里

静态链接库和动态链接库的区别

```
jkx@ubuntu:~/work/chapt6/myfun$ ls -l main_*
-rwxrwxr-x 1 jkx jkx 7356 Oct 18 21:09 main_dynam
-rwxrwxr-x 1 jkx jkx 7432 Oct 18 21:08 main_stati
```

```
jkx@ubuntu:~/work/chapt6/myfun$ rm libmyfun.a
jkx@ubuntu:~/work/chapt6/myfun$ ./main_static
the result is 3
jkx@ubuntu:~/work/chapt6/myfun$ rm libmyfun.so
jkx@ubuntu:~/work/chapt6/myfun$ ./main_dynamic
./main_dynamic: error while loading shared libraries: libmyfun.so:
hared object file: No such file or directory
```

- 大小有别，运行时的依赖有别

Linux C编程

- GCC简介
- GCC的使用
- GCC编译举例
- 程序调试：gdb
- 程序维护：Makefile文件及make

gdb调试

- gdb是Linux系统中一个功能强大的GNU调试程序，它可以调试C和C++程序，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提供如下功能：
  - （1）运行程序，设置断点。
  - （2）当程序停止时，可以检查程序的状态。
  - （3）动态监视程序中变量的值，可以单步逐行执行代码，观察程序的运行状态。
  - （4）分析崩溃程序产生的core文件

gdb调试switch.c

```
#include <stdio.h>
int main(void)
{
   int sum = 0, i = 0;
   char input[5];
   while (1)
    {
       printf("please input a int:\n");
       scanf("%s", input);
       for (i = 0; input[i] != '\0'; i++)
            sum += input[i] - '0';
       printf("input=%d\n", sum);
     }
    return 0;
}
```

从键盘输入一串数字字符串放到**input**中，计算出这些数字之和，比如

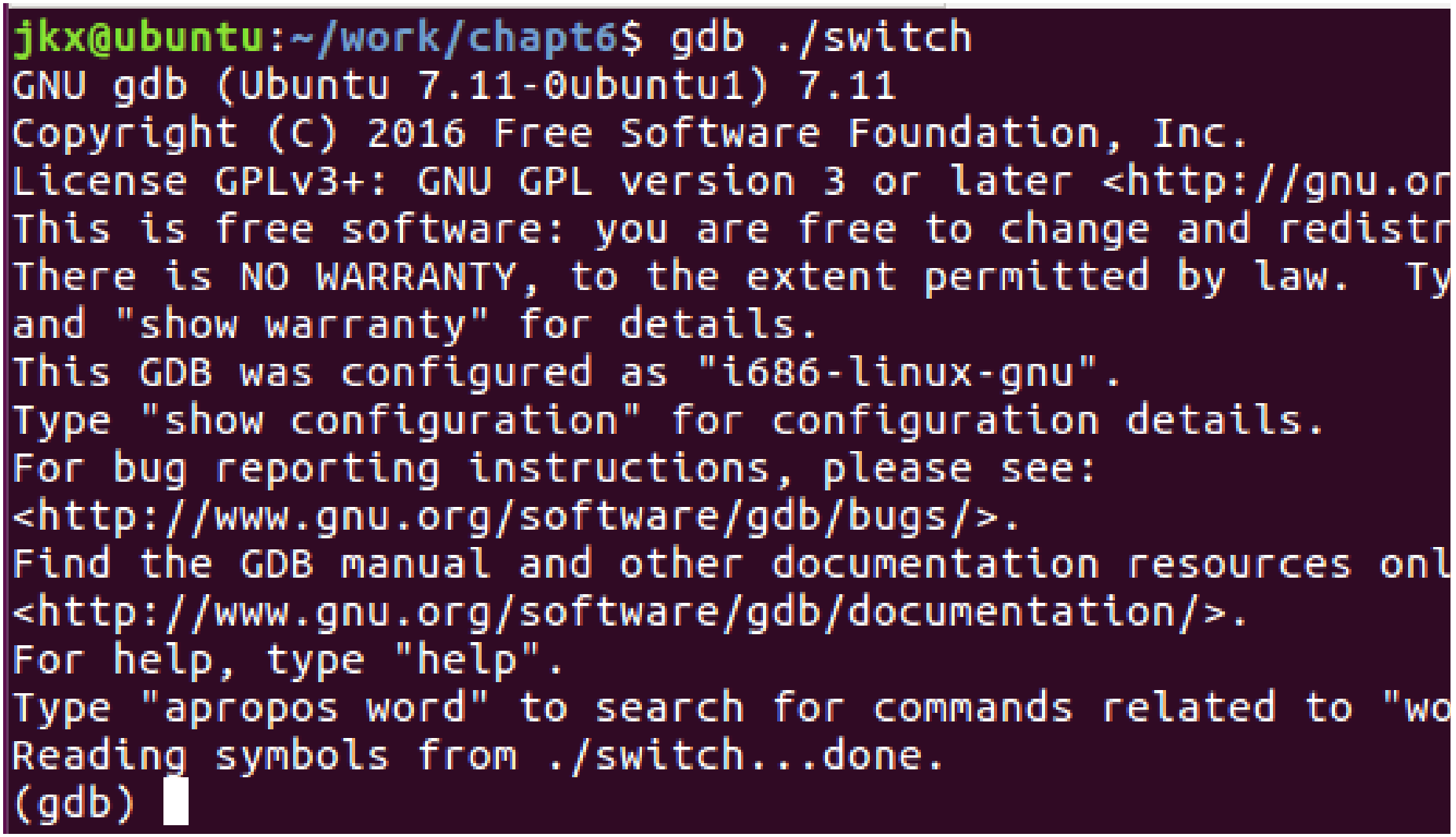
输入**12345**，输出**15**

gdb调试switch.c

注意：没有-g
不能调试

```
kx@ubuntu:~/work/chapt6$ gcc -o switch -g switch.c
```

```
jkx@ubuntu:~/work/chapt6$ ./switch
please input a int:
234
input=9
please input a int:
89
input=26
please input a int:
```


jkx@ubuntu:~/work/chapt6$ gdb ./switch
GNU gdb (Ubuntu 7.11-0ubuntu1) 7.11
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.or
This is free software: you are free to change and redistr
There is NO WARRANTY, to the extent permitted by law. Ty
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources onl
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "wo
Reading symbols from ./switch...done.
(gdb)

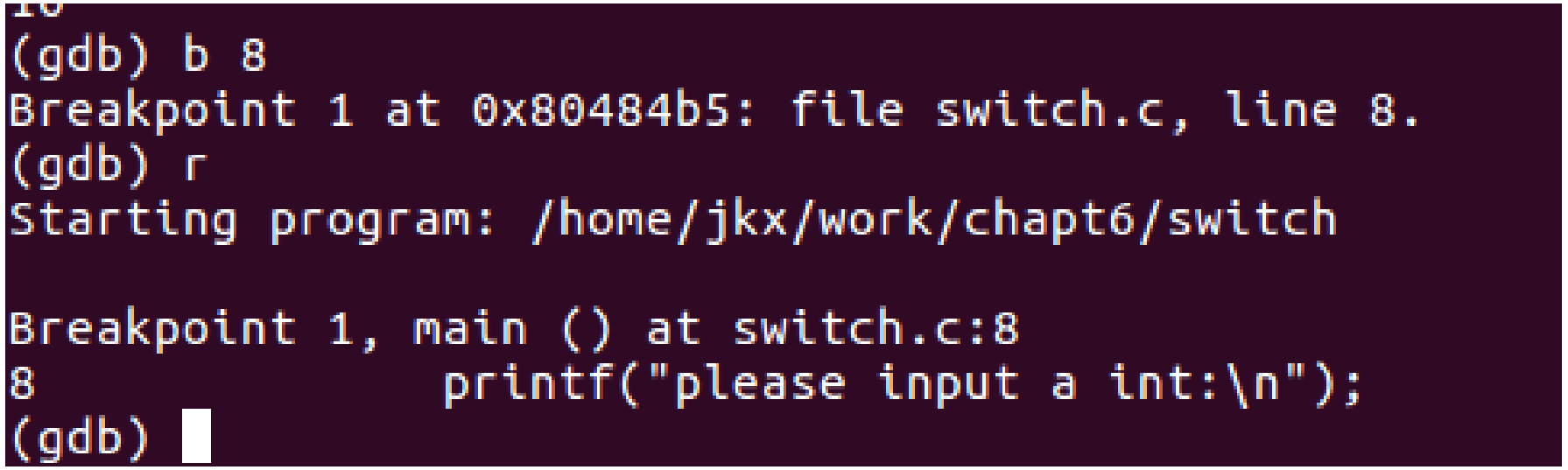
gdb的命令

- （1）用**list**命令（简写**l**）列出源代码及行号
- （2）用**break**命令（简写为**b**）在怀疑行设一个断点（**Breakpoint**）。
- （3）用**run**命令（简写**r**）运行程序，程序停在断点处。
- （4）用**print**（简写**p**）打印可疑变量值
- （5）用**next**（简写**n**）执行一条语句，如果碰到函数可以用**step**（简写**s**）进入函数内部
- （6）用**continue**命令（简写**c**）继续执行程序
- （7）用**quit**（简写**q**）结束

list命令

```
(gdb) l
1       #include <stdio.h>
2       int main(void)
3       {
4           int sum = 0, i = 0;
5           char input[5];
6           while (1)
7            {
8               printf("please input a int:\n");
9               scanf("%s", input);
10              for (i = 0; input[i] != '\0'; i++)
(gdb) l
11                  sum += input[i] - '0';
12              printf("input=%d\n", sum);
13           }
14          return 0;
15      }
16
```

break和run命令



print和next命令

```
(gdb) n
please input a int:
9                   scanf("%s", input);
(gdb) n
234
10                  for (i = 0; input[i] != '\0'; i++)
(gdb) p input
$1 = "234\000\277"
(gdb) p input[0]
$2 = 50 '2'
(gdb) p i
$3 = 0
(gdb) n
11                      sum += input[i] - '0';
(gdb) n
10                  for (i = 0; input[i] != '\0'; i++)
(gdb) p sum
$4 = 2
```

continue命令

```
10              for (i = 0; input[i] != '\0'; i++)
(gdb) p sum
$6 = 9
(gdb) p i
$7 = 2
(gdb) c
Continuing.
input=9

Breakpoint 1, main () at switch.c:8
8               printf("please input a int:\n");
(gdb) p
$8 = 2
(gdb) n
please input a int:
9               scanf("%s", input);
(gdb) 89
Undefined command: "89".  Try "help".
(gdb) n
89
10              for (i = 0; input[i] != '\0'; i++)
(gdb) p sum
$9 = 9
```

quit命令

```
(gdb) p sum
$9 = 9
(gdb) q
A debugging session is active.

        Inferior 1 [process 4001] will be killed.

Quit anyway? (y or n) y
```

Linux C编程

- GCC简介
- GCC的使用
- GCC编译举例
- 程序调试：gdb
- 程序维护：Makefile文件及make

Makefile文件及make命令

- ✓ 在开发大系统时，经常要将程序划分为许多模块。各个模块之间存在着各种各样的依赖关系，在Linux中通常使用 Makefile来管理。
- ✓ 由于各个模块间不可避免存在关联，所以当一个模块改动后，其他模块也许会有所更新，为此，在Linux系统中，专门提供了一个make工具来自动维护目标文件。
- ✓ make工具可以针对项目中的每个源文件设置编译、连接参数，实现项目的自动编译。

Makefile 文件

- ✓ Makefile文件的内容是描述项目或软件（包）中的模块之间的相互依赖关系以及目标文件、如何生成可执行文件等。
- ✓ Makefile文件可认为是一个工程规划文件。
- ✓ Makefile时由规则来组成的,每一条规则都有三部分组成:目标(object),依赖(dependency)和命令(command)。

```
target: dependence file1  file2 ……
        command1
        command2
```

注意这里是一个**Tab**

将命令作用在依赖文件上

产生目标

一个简单Makefile例子

- 包含三个.c文件和两个.h文件

main.c

```
#include "mytool1.h"
#include "mytool2.h"
int main(int argc,char **argv)
{
  mytool1_print("hello");
  mytool2_print("hello");
 }
```

mytool1.c

```
#include "mytool1.h"
#include <stdio.h>
void mytool1_print(char
*print_str)
{   printf("This is mytool1 print
%s\n",print_str); }
```

mytool2.c

```
#include "mytool2.h"
#include <stdio.h>
void mytool1_print(char
*print_str)
{   printf("This is mytool2 print
%s\n",print_str); }
```

一个简单Makefile例子

- 包含三个.c文件和两个.h文件

mytool1.h

```
#ifndef  _MYTOOL_1_H
#define  _MYTOOL_1_H
void mytool1_print(char
*print_str);
#endif
```

mytool1.h

```
#ifndef  _MYTOOL_1_H
#define  _MYTOOL_1_H
void mytool1_print(char
*print_str);
#endif
```

Makefile文件

```
main:main.o mytool1.o mytool2.o
	gcc -o main main.o mytool1.o mytool2.o
main.o:main.c mytool1.h mytool2.h
	gcc -c main.c
mytool1.o:mytool1.c mytool1.h
	gcc -c mytool1.c
mytool2.o:mytool2.c mytool2.h
	gcc -c mytool2.c
clean:
	rm -f *.o main
```

Makefile 的宏(macro)定义

make中使用宏，要先定义，然后在makefile中引用。

宏的定义格式：　宏名 = 宏的值（宏名一般习惯用大写字母）

宏的引用：　$(宏名)

```
GCC=gcc
OBJS=main.o mytool1.o mytool2.o
main:$(OBJS)
    $(GCC) -o main $(OBJS)
main.o:main.c mytool1.h mytool2.h
    $(GCC) -c main.c
mytool1.o:mytool1.c mytool1.h
    $(GCC) -c mytool1.c
mytool2.o:mytool2.c mytool2.h
    $(GCC) -c mytool2.c
```

Makefile 的简化

```
GCC=gcc
OBJS=main.o mytool1.o mytool2.o
CFLAG=-g –Wall –werror
main:$(OBJS)
    $(GCC) -o $@ $^ $(CFLAG)
main.o:main.c mytool1.h mytool2.h
    $(GCC) -c $<  $(CFLAG)
mytool1.o:mytool1.c mytool1.h
    $(GCC) -c $< $(CFLAG)
mytool2.o:mytool2.c mytool2.h
    $(GCC) -c $< $(CFLAG)
clean:
    rm main $(OBJS)
```

$@ 目标文件

$^ 所有的依赖文件

$< 第一个依赖文件

课堂作业

- 以下关于GCC选项说法错误的是：
  - A、-c 只编译并生成目标文件。
  - B、-w生成警告信息。
  - C、-g 生成调试信息。
  - D、-o FILE生成指定的输出文件。

- make命令的用途是：
  - A、 清空/tmp目录
  - B、编译
  - C、安装软件
  - D、 什么都不作

作业

- P230/ 5，6
- 附加题：
  - 当前目录是/home/jkx/work，里面有一个名为test.c的文件，test.c依赖位于/home/jkx/includes目录下的头文件defs.h，和位于/home/jkx/libs目录下的库文件libmy.so。请写出将test.c生成可执行文件test的gcc命令。