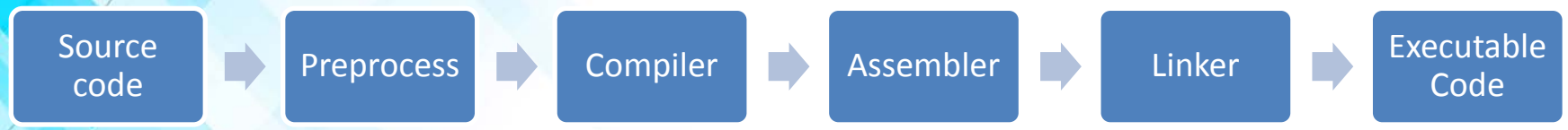
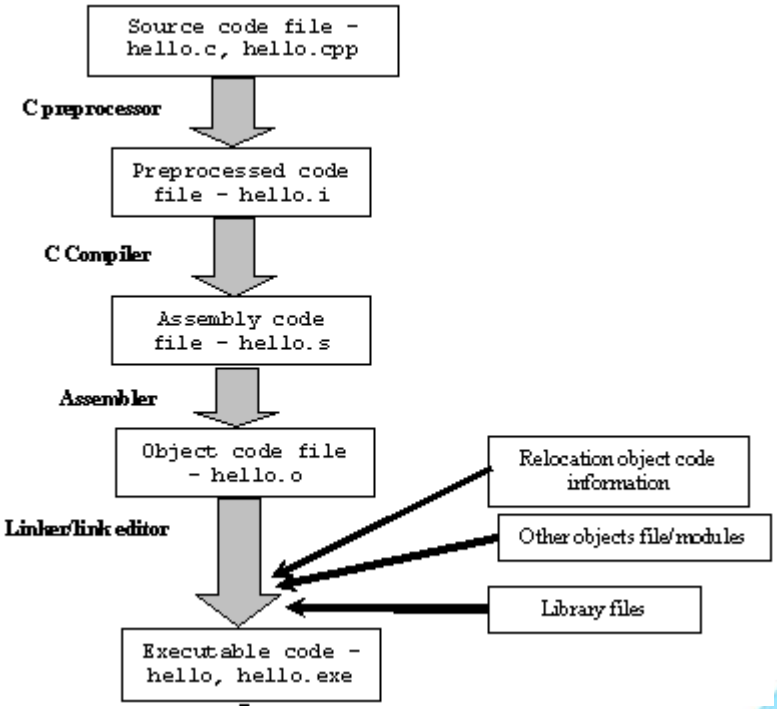
Experiment 1 How to compile and debug a program in Linux

信息与软件工程学院
School of information and software engineering

Program Compilation Process





Command for compiling a program

Proprocess: gcc -E hello.c -o hello.i

Compiler: $gcc –S hello.i –o hello.s

Assembly: $ gcc –c hello.c –o hello.o

GCC

The GNU Compiler Collection includes front ends for C, C++, Objective-C, Fortran, Java, and Go, as well as libraries for these languages. GCC was originally written as the compiler for the GNU operating system.

GCC includes:

- Cpp(preprocess)
- Gcc (c compiler), g++(c++ compiler) and so on;
- the binary tool like the binutils
  - Assembler (as)
  - Link editor (ld)

gcc

Gcc/g++ both can compile c code or c++ code.

- G++ regard all files as the c++ source code, whatever it is a c code or c++ code.
- Gcc exactly distinguish the file through its suffix, .c is the c code and .cpp is the c++ code.
- Gcc can't automatically link the c++ library. (it needs the argument –lstdc++)

GCC

Gcc command:

Gcc [option] <filename>

- -o filename: the output file is the filename. This option don't care which output will be produced, the output can be executable file, object file, assembly file or the preprocessed code.
- Without the -o option, the default output is the executable file "a.out", the object file "sourcename.o", the assembly file "sourcename.s" and the preprocessed code will be sent to the stdout.

Aim to the source code main.c, after execute the following command, it will be compiled as the final executable file (in default, this process includes the preprocessing, compiling, assembling and linking)

gcc main.c –o main

Other option in gcc

-d: Marcos definition. Equal to the #define MACRO in the source code, but this marcos is used for all source files.

#define PI 3.14159(the value of PI is the 3.14159 in source code)

gcc -DPI=3.14159 main.c

-I: the searching path for the head file. If the user's head file is not in the searching path, this option is used to specify the extra searching path.

gcc helloworld.c –I /usr/include –o helloworld (jion the /usr/include into the searching path of the head file)

Other options in gcc

-w: forbidden all warning information.

-Wall: open all warning options. Outputting the warning information.

For warning information, it don't represent there are some problems in the source code. In most way, it mean there is a risk in the source code. But some times, it represents code error.

Suggesting to open -Wall, to get the warning information for the source code.

Other options in gcc

-g: produce the debug information. GDB can use these debugging information.

-ggdb: it can get more rich debugging information. But the executable file can't be debugged by other debugger.

When employing the debug, the size of binary file will increase sharply.

GDB

GDB, the GNU Project debugger, allows you to see what is going on `inside' another program while it executes -- or what another program was doing at the moment it crashed.

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

GDB Starting

Compile the program and execute the program.

% gcc -g <program> -o filename

% ./filename

The -g option is important because it enables meaningful GDB debugging.

Start the debugger:

gdb <program>

This only starts the debugger; it does not start running the program in the debugger.

Debugging a program that produces a core dump:

gdb program core

Debugging a running process

gdb program <processid>

Basic commands in GDB

help [command], to check how to use the command. If just input the help, the classification of command will be listed. Then we can employ help command to check all command under different classification.

Exiting the gdb by adopting the quit command or just press "ctrl+d".

Break Command

To set a break point in the source code.

Syntax: break [LOCATION] [thread THREADNUM] [if CONDITION]

[LOCATION]:

- linenum
- function name
- filename:linenum
- filename:function
- class:function（c++）
- Example: b 123; b main; b increase:main; b increase:123

[thread THREADNUM]:

- When debugging the multithreading program, switching to which thread or in which thread to set the breakpoint
- Example:break frik.c:13 thread 28

Break Command

[if CONDITION]:

When the condition is true, the breakpoint will be come into force. It also named as the condition breakpoint.

Example: b 123 if index==2

When the index equal to 2. the program will be stoped.

Watchpoint

You can use a watchpoint to stop execution whenever the value of an expression changes, without having to predict a particular place where this may happen.

watch <expr>: Set a watchpoint for an expression. gdb will break when the expression expr is written into by the program and its value changes. The simplest (and the most popular) use of this command is to watch the value of a single variable: "(gdb) watch foo".

rwatch <expr>: Set a watchpoint that will break when the value of expr is read by the program.

awatch <expr> : Set a watchpoint that will break when expr is either read from or written into by the program.

Info watchpoints: his command prints a list of watchpoints, using the same format as info break

Clear the breakpoints and watchpoints

clear [linenum] [function name] : clear all breakpoints, but not for the watchpoints.

delete <num>: clear the breakpoint or watchpoint whose number is num.

disable <num>: disable the breakpoint.

enable <num>: enable the breakpoint.

Examining data

The usual way to examine data in your program is with the print command (abbreviated p), or its synonym inspect. It evaluates and prints the value of an expression of the language your program is written in.

Syntax: print expr or print /f expr

expr is an expression (in the source language). By default the value of expr is printed in a format appropriate to its data type; you can choose a different format by specifying `/f', where f is a letter specifying the format;

Example:

Examining the value of i：(gdb)p i

Examining the value of i*5：(gdb)p i*5

Examining the value of name in the s strcuture：(gdb)p s.name

Other related command：undisplay，delete display，disable display，enable display，info display

Examining Data

If you find that you want to print the value of an expression frequently (to see how it changes), you might want to add it to the automatic display list so that GDB prints its value each time your program stops. The command used for this purpose is the "display".

display *expr*

Add the expression *expr* to the list of expressions to display each time your program stops.

display/*fmt expr*

For *fmt* specifying only a display format and not a size or count, add the expression *expr* to the auto-display list but arrange to display it each time in the specified format *fmt*

Examining the source files

To print lines from a source file, use the list command (abbreviated l). By default, ten lines are printed.

List linenum

Print lines centered around line number linenum in the current source file.

+offset

Specifies the line offset lines after the last line printed.

-offset

Specifies the line offset lines before the last line printed.

filename:number

Specifies line number in the source file filename.

filename:function

Specifies the line of the open-brace that begins the body of the function in the file *filename*. You only need the file name with a function name to avoid ambiguity when there are identically named functions in different source files.

Command for debugging

Step: Continue running your program until control reaches a different source line, then stop it and return control to gdb. This command is abbreviated s.

Next: Continue to the next source line in the current (innermost) stack frame. This is similar to step, but function calls that appear within the line of code are executed without stopping. Execution stops when control reaches a different line of code at the original stack level that was executing when you gave the next command. This command is abbreviated n.

Finish: Continue running until just after function in the selected stack frame returns. Print the returned value (if any). This command can be abbreviated as fin.

Basic Command

Dir: When you start gdb, its source path includes only 'cdir' and 'cwd', in that order. To add other directories, use the directory command. This command will add directory *dirname* to the front of the source path.

R[[arg 1][arg 2]….]: Use the run command to start your program under gdb. These arguments are required by the program.

Info [**]: examining the breakpoint, function name, class name and so on.

info b: printing a list of breakpoints.

Experiment

Requirement:

C language programming: to define a list based on the data structure, and to design algorithms for creating a list, inserting a node and printing a list;

Compiling the source file and then debugging in the Ubuntu.

The screenshot of compiling and debugging are needed for your report.

Experiment

Predefined data structure:

```
typedef struct stuInfo {
        char  stuName[10];
        int    Age
} ElemType
typedef struct node {
        Elemtype  data;
        struct  node  *next;
} ListNode, *ListPtr;
```

Main() Function

```
int main(int argc,char argv[])
{
        while(1)
        {
                printf("1 Create List\n");
                printf("2 Printf List\n");
                printf("3 Insert List\n");
                printf("4 Quit\n");
                char command =getchar();
                switch(command)
                {
                        case '1':ListHead = CreateList();
                        break;
                        case '2':PrintList(ListHead);
                        break;
                        case '3':InsertList(ListHead);
                        break;
                        case '4':
                        return 0;
                }
        }
        return 0;
}
```

Functions

Use the standard I/O Library for this experiment:

For createlist():

FILE*pfile;

FILE * fopen(const char * path,const char * mode);

int fclose(FILE *fp);

int feof(FILE *fp);

int fscanf(FILE *restrict fp, const char *restrict format, ...);

(fscanf(pfile, "%s\t", node->data.stuName))

For InsertList():

int fflush(FILE *fp);

Other:

ListPtr head = (ListPtr)malloc(sizeof(ListNode));