Programming Background

2

Abstract

This chapter covers the background information needed for systems programming. It introduces several GUI based text editors, such as vim, gedit and EMACS, to allow readers to edit files. It shows how to use the EMACS editor in both command and GUI mode to edit, compile and execute C programs. It explains program development steps. These include the compile-link steps of GCC, static and dynamic linking, format and contents of binary executable files, program execution and termination. It explains function call conventions and run-time stack usage in detail. These include parameter passing, local variables and stack frames. It also shows how to link C programs with assembly code. It covers the GNU make facility and shows how to write Makefiles by examples. It shows how to use the GDB debugger to debug C programs. It points out the common errors in C programs and suggests ways to prevent such errors during program development. Then it covers advanced programming techniques. It describes structures and pointer in C. It covers link lists and list processing by detailed examples. It covers binary trees and tree traversal algorithms. The chapter cumulates with a programming project, which is for the reader to implement a binary tree to simulate operations in the Unix/Linux file system tree. The project starts with a single root directory node. It supports mkdir, rmdir, creat, rm, cd, pwd, ls operations, saving the file system tree as a file and restoring the file system tree from saved file. The project allows the reader to apply and practice the programming techniques of tokenizing strings, parsing user commands and using function pointers to invoke functions for command processing.

2.1 Text Editors in Linux

2.1.1 Vim

Vim (Linux Vi and Vim Editor 2017) is the standard built-in editor of Linux. It is an improved version of the original default Vi editor of Unix. Unlike most other editors, vim has 3 different operating modes, which are

. **Command mode:** for entering commands
. **Insert mode:** for entering and editing text
. Last-line mode: for saving files and exit

K. C. Wang, *Systems Programming in Unix/Linux*, https://doi.org/10.1007/978-3-319-92429-8_2

When vim starts, it is in the default **Command mode**, in which most keys denote special commands. Examples of command keys for moving the cursor are

h: move cursor one char to the left;	**l: move cursor one char to the right**
j: move cursor down one line;	**k: move cursor up one line**

When using vim inside X-window, cursor movements can also be done by arrow keys. To enter text for editing, the user must switch vim into **Insert mode** by entering either the **i** (insert) or **a** (append) command:

i: switch to Insert mode to insert text;
a: switch to Insert mode to append text

To exit Insert mode, press the **ESC key** one or more times. While in Command mode, enter the **:** key to enter the **Last-line mode**, which is for saving texts as files or exit vim:

:w write (save) file
:q exit vim
:wq save and exit
:q! force exit without saving changes

Although many Unix users are used to the different operating modes of vim, others may find it somewhat unnatural and inconvenient to use as compared with other Graphic User Interface **(GUI)** based editors. The following editors belong to what is commonly known as What You See Is What You Get **(WYSIWYG)** type of editors. In a WYSIWYG editor, a user may enter text, move the cursor by arrow keys, just like in regular typing. Commands are usually formed by entering a special **meta key**, together with, or followed by, a letter key sequence. For example,

Control-C: abort or exit,
Control-K: delete line into a buffer,
Control-Y: yank or paste from buffer contents
Control-S: save edited text, etc.

Since there is no mode switching necessary, most users, especially beginners, prefer WYSUWYG type editors over vim.

2.1.2 Gedit

Gedit is the default text editor of the **GNOME** desktop environment. It is the default editor of Ubuntu, as well as other Linux that uses the GNOME GUI user interface. It includes tools for editing both source code and structured text such as markup languages.

2.1.3 Emacs

Emacs (GNU Emacs 2015) is a powerful text editor, which runs on many different platforms. The most popular version of Emacs is **GNU Emacs**, which is available in most Linux distributions.

All the above editors support direct inputs and editing of text in full screen mode. They also support search by keywords and replace strings by new text. To use these editors, the user only needs to learn a few basics, such as how to start up the editor, input text for editing, save edited texts as files and then exit the editor.

Depending on the Unix/Linux distribution, some of the editors may not be installed by default. For example, **Ubuntu** Linux usually comes with **gedit, nano** and **vim**, but not **emacs**. One nice feature of Ubuntu is that, when a user tries to run a command that is not installed, it will remind the user to install it. As an example, if a user enters

```
emacs filename
```

Ubuntu will display a message saying "The program emacs is currently not installed. You can install it by typing apt-get install emacs". Similarly, the user may install other missing software packages by the **apt-get** command.

2.2 Use Text Editors

All text editors are designed to perform the same task, which is to allow users to input texts, edit them and save the edited texts as files. As noted above, there are many different text editors. Which text editor to use is a matter of personal preference. Most Linux users seem to prefer either **gedit** or **emacs** due to their GUI interface and ease to use. Between the two, we strongly recommend emacs. The following shows some simple examples of using emacs to create text files.

2.2.1 Use Emacs

First, from a pseudo terminal of X-windows, enter the command line

```
emacs [FILENAME]                # [ ] means optonal
```

to invoke the emacs editor with an optional file name, e.g. t.c. This will start up emacs in a separate window as show in Fig. 2.1. On top of the emacs window is a menu bar, each of which can be opened to show additional commands, which the user can invoke by clicking on the menu icons. To begin with, we shall not use the menu bar and focus on the simple task of creating a C program source file. When emacs starts, if the file t.c already exists, it will open the file and load its contents into a buffer for editing. Otherwise, it will show an empty buffer, ready for user inputs. Fig. 2.1 shows the user input lines. Emacs recognizes any .c file as source file for C programs. It will indent the lines in accordance with the C code line conventions, e.g. it will match each left { with a right }with proper indentations automatically. In fact, it can even detect incomplete C statements and show improper indentations to alert the user of possible syntax errors in the C source lines.

After creating a source file, enter the meta key sequence **Control-x-c** to save the file and exit. If the buffer contains modified and unsaved text, it will prompt the user to save file, as shown on the bottom line of Fig. 2.2. Entering y will save the file and exit emacs. Alternatively, the user may also click the **Save icon** on the menu bar to save and exit.

Fig. 2.1 Use emacs 1

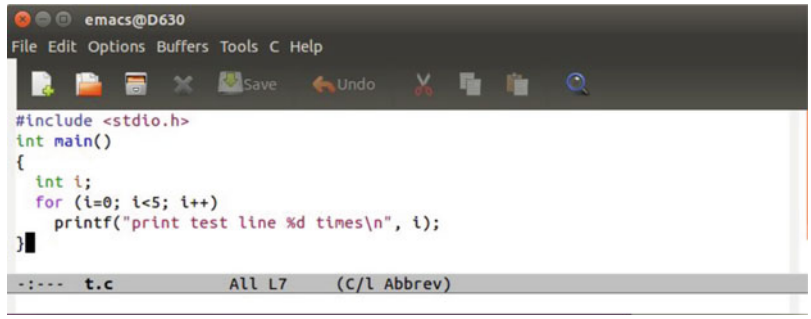
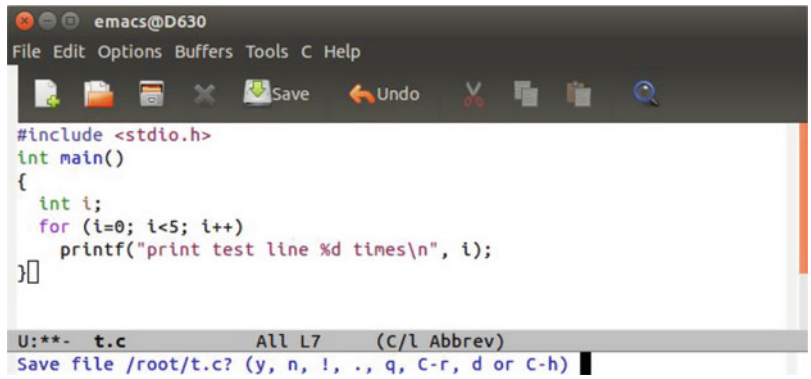


Fig. 2.2 Use emacs 2



2.2.2 Emacs Menus

At the top of the emacs window is a menu bar, which includes the icons

`File Edit Options Buffers Tools C Help`

The **File** menu supports the operations of open file, insert file and save files. It also supports printing the editing buffer, open new windows and new frames.
The **Edit** menu supports search and replace operations.
The **Options** menu supports functions to configure emacs operations.
The **Buffers** menu supports selection and display of buffers.
The **Tools** menu supports compilation of source code, execution of binary executables and debugging.
The **C** menu supports customized editing for C source code.
The **Help** menu provides support for emacs usage, such as a simple emacs tutorial.

As usual, clicking on a menu item will display a table of submenus, allowing the user to choose individual operations. Instead of commands, the user may also use the emacs menu bar to do text editing, such as undo last operation, cut and past, save file and exit, etc.

2.2.3 IDE of Emacs

Emacs is more than a text editor. It provides an Integrated Development Environment **(IDE)** for software development, which include compile C programs, run executable images and debugging

program executions by **GDB**. We shall illustrate the IDE capability of emacs in the next section on program development.

2.3 Program Development

2.3.1 Program Development Steps

The steps of developing an executable program are as follows.

(1). Create source files: Use a text editor, such as gedit or emacs, to create one or more source files of a program. In systems programming, the most important programming languages are C and assembly. We begin with C programs first.

Standard comment lines in C comprises matched pairs of /* and */. In addition to the standard comments, we shall also use // to denote comment lines in C code for convenience. Assume that t1.c and t2.c are the source files of a C program.

```
/********************** t1.c file ****************************/
int g = 100;                  // initialized global variable
int h;                        // uninitialized global variable
static int s;                 // static global variable

main(int argc, char *argv[ ]) // main function
{
  int a = 1; int b;          // automatic local variables
  static int c = 3;          // static local variable
  b = 2;
  c = mysum(a,b);             // call mysum(), passing a, b
  printf("sum=%d\n", c);      // call printf()
}
/********************** t2.c file ****************************/
extern int g;                 // extern global variable
int mysum(int x, int y)       // function heading
{
   return x + y + g;
}
```

2.3.2 Variables in C

Variables in C programs can be classified as **global, local, static, automatic** and **registers**, etc. as shown in Fig. 2.3.

Global variables are defined outside of any function. **Local variables** are defined inside functions. Global variables are unique and have only one copy. **Static globals** are visible only to the file in which they are defined. **Non-static globals** are visible to all the files of the same program. Global variables can be initialized or uninitialized. Initialized globals are assigned values at compile time. Uninitialized globals are cleared to 0 when the program execution starts. Local variables are visible only to the function in which they are defined. By default, local variables are **automatic**; they come into existence

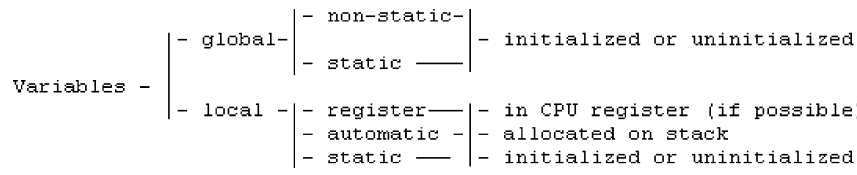


Fig. 2.3 Variables in C

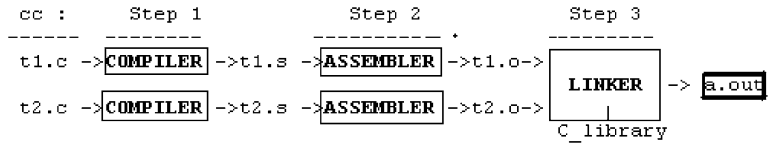


Fig. 2.4 Program development steps

when the function is entered and they logically disappear when the function exits. For **register variables**, the compiler tries to allocate them in CPU registers. Since automatic local variables do not have any allocated memory space until the function is entered, they cannot be initialized at compile time. **Static local variables** are permanent and unique, which can be initialized. In addition, C also supports **volatile variables**, which are used as memory-mapped I/O locations or global variables that are accessed by interrupt handlers or multiple execution threads. The volatile keyword prevents the C compiler from optimizing the code that operates on such variables.

In the above t1.c file, g is an initialized global, h is an uninitialized global and s is a static global. Both g and h are visible to the entire program but s is visible only in the t1.c file. So t2.c can reference g by declaring it as extern, but it cannot reference s because s is visible only in t1.c. In the main() function, the local variables a, b are automatic and c is static. Although the local variable a is defined as int a = 1, this is not an initialization because a does not yet exist at compile time. The generated code will assign the value 1 to the current copy of a when main() is actually entered.

2.3.3 Compile-Link in GCC

(2). Use gcc to convert the source files into a binary executable, as in

```
gcc  t1.c  t2.c
```

which generates a binary executable file named a.out. In Linux, cc is linked to gcc, so they are the same.

(3). What's gcc? **gcc** is a program, which consists of three major steps, as shown in Fig. 2.4.

Step 1. Convert C source files to assembly code files: The first step of cc is to invoke the C COMPILER, which translates .c files into .s files containing **assembly code** of the target machine. The C compiler itself has several phases, such as preprocessing, lexical analysis, parsing and code generations, etc, but the reader may ignore such details here.

Step 2. Convert assembly Code to **OBJECT code**: Every computer has its own set of machine instructions. Users may write programs in an assembly language for a specific machine. An ASSEMBLER is a program, which translates assembly code into machine code in binary form. The resulting .o files are called OBJECT code. The second step of cc is to invoke the ASSEMBLER to translate .s files to .o files. Each .o file consists of

. a header containing sizes of CODE, DATA and BSS sections
. a CODE section containing machine instructions
. a DATA section containing initialized global and initialized static local variables
. a BSS section containing uninitialized global and uninitialized static local variables
. relocation information for pointers in CODE and offsets in DATA and BSS
. a Symbol Table containing non-static globals, function names and their attributes.

Step 3: LINKING: A program may consist of several .o files, which are dependent on one another. In addition, the .o files may call C library functions, e.g. printf(), which are not present in the source files. The last step of cc is to invoke the LINKER, which combines all the .o files and the needed library functions into a single binary executable file. More specifically, the LINKER does the following:

. Combine all the CODE sections of the .o files into a single Code section. For C programs, the combined Code section begins with the default C startup code **crt0.o**, which calls main(). This is why every C program must have a unique main() function.
. Combine all the DATA sections into a single Data section. The combined Data section contains only initialized globals and initialized static locals.
. Combine all the BSS sections into a single bss section.
. Use the relocation information in the .o files to adjust pointers in the combined Code section and offsets in the combined Data and bss sections.
. Use the Symbol Tables to resolve cross references among the individual .o files. For instance, when the compiler sees c = mysum(a, b) in t1.c, it does not know where mysum is. So it leaves a blank (0) in t1.o as the entry address of mysum but records in the symbol table that the blank must be replaced with the entry address of mysum. When the linker puts t1.o and t2.o together, it knows where mysum is in the combined Code section. It simply replaces the blank in t1.o with the entry address of mysum. Similarly for other cross referenced symbols. Since static globals are not in the symbol table, they are unavailable to the linker. Any attempt to reference static globals from different files will generate a cross reference error. Similarly, if the .o files refer to any undefined symbols or function names, the linker will also generate cross reference errors. If all the cross references can be resolved successfully, the linker writes the resulting combined file as a.out, which is the binary executable file.

2.3.4 Static vs. Dynamic Linking

There are two ways to create a binary executable, known as **static linking** and **dynamic linking.** In static linking, which uses a **static library**, the linker includes all the needed library function code and data into a.out. This makes a.out complete and self-contained but usually very large. In dynamic linking, which uses a **shared library**, the library functions are not included in a.out but calls to such functions are recorded in a.out as directives. When execute a dynamically linked a.out file, the operating system loads both a.out and the shared library into memory and makes the loaded library code accessible to a.out during execution. The main advantages of dynamic linking are:

. The size of every a.out is reduced.
. Many executing programs may share the same library functions in memory.
. Modifying library functions does not need to re-compile the source files again.

Libraries used for dynamic linking are known as **Dynamic Linking Libraries (DLLs)**. They are called **Shared Libraries** (.so files) in Linux. Dynamically loaded (DL) libraries are shared libraries which are loaded only when they are needed. DL libraries are useful as plug-ins and dynamically loaded modules.

2.3.5 Executable File Format

Although the default binary executable is named a.out, the actual file format may vary. Most C compilers and linkers can generate executable files in several different formats, which include

(1) **Flat binary executable**: A flat binary executable file consists only of executable code and initialized data. It is intended to be loaded into memory in its entirety for execution directly. For example, bootable operating system images are usually flat binary executables, which simplifies the boot-loader.
(2) **a.out executable file**: A traditional a.out file consists of a header, followed by code, data and bss sections. Details of the a.out file format will be shown in the next section.
(3) **ELF executable file**: An Executable and Linking Format **(ELF)** (Youngdale 1995) file consists of one or more program sections. Each program section can be loaded to a specific memory address. In Linux, the default binary executables are ELF files, which are better suited to dynamic linking.

2.3.6 Contents of a.out File

For the sake of simplicity, we consider the traditional a.out files first. ELF executables will be covered in later chapters. An a.out file consists of the following sections:

(1) header: the header contains loading information and sizes of the a.out file, where
tsize = size of Code section;
dsize = size of Data section containing initialized globals and static locals;
bsize = size of bss section containing uninitialized globals and static locals;
total_size = total size of a.out to load.
(2) **Code Section**: also called the **Text section**, which contains executable code of the program. It begins with the standard C startup code **crt0.o**, which calls main().
(3) **Data Section**: The Data section contains initialized global and static data.
(4) **Symbol table**: optional, needed only for run-time debugging.

Note that the **bss section**, which contains uninitialized global and static local variables, is not in the a.out file. Only its size is recorded in the a.out file header. Also, automatic local variables are not in a.out. Figure 2.5 shows the layout of an a.out file.

In Fig. 2.5, _brk is a symbolic mark indicating the end of the bss section. The total loading size of a.out is usually equal to _brk, i.e. equal to tsize+dsize+bsize. If desired, _brk can be set to a higher value for a larger loading size. The extra memory space above the bss section is the **HEAP** area for dynamic memory allocation during execution.

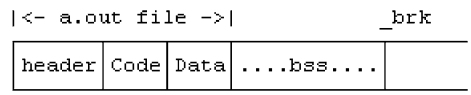


Fig. 2.5 Contents of a.out file

2.3.7 Program Execution

Under a Unix-like operating system, the **sh** command line

```
a.out one two three
```

executes a.out with the token strings as **command-line parameters**. To execute the command, sh forks a child process and waits for the child to terminate. When the child process runs, it uses a.out to create a new execution image by the following steps.

(1) Read the header of a.out to determine the total memory size needed, which includes the size of a stack space:

```
TotalSize = _brk + stackSize
```

where stackSize is usually a default value chosen by the OS kernel for the program to start. There is no way of knowing how much stack space a program will ever need. For example, the trivial C program

```
main(){ main(); }
```

will generate a segmentation fault due to **stack overflow** on any computer. So the usual approach of an OS kernel is to use a default initial stack size for the program to start and tries to deal with possible stack overflow later during run-time.

(2) It allocates a memory area of TotalSize for the execution image. Conceptually, we may assume that the allocated memory area is a single piece of contiguous memory. It loads the Code and Data sections of a.out into the memory area, with the stack area at the high address end. It clears the bss section to 0, so that all uninitialized globals and static locals begin with the initial value 0. During execution, the stack grows downward toward low address.
(3) Then it abandons the old image and begins to execute the new image, which is shown in Fig. 2.6.

In Fig. 2.6, _brk at the end of the bss section is the program's initial "break" mark and _splimit is the stack size limit. The Heap area between bss and Stack is used by the C library functions malloc()/free() for dynamic memory allocation in the execution image. When a.out is first loaded, _brk and _splimit may coincide, so that the initial Heap size is zero. During execution, the process may use the brk (address) or sbrk(size) system call to change _brk to a higher address, thereby increasing the Heap size. Alternatively, malloc() may call brk() or sbrk() implicitly to expand the Heap size. During execution, a stack overflow occurs if the program tries to extend the stack pointer below _splimit. On machines with memory protection, this will be detected by the memory management hardware as an error, which traps the process to the OS kernel. Subject to a maximal size limit, the OS kernel may grow the stack by allocating additional memory in the process address space, allowing the execution to continue. A stack

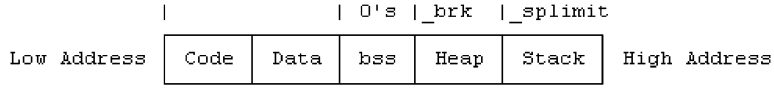


Fig. 2.6 Execution image

overflow is fatal if the stack cannot be grown any further. On machines without suitable hardware support, detecting and handling stack overflow must be implement in software.

(4). Execution begins from crt0.o, which calls main(), passing as parameters argc and argv to main(), which can be written as

```
int main( int argc, char *argv[ ] ) {  ... .  }
```

where argc = number of command line parameters and each argv[ ] entry points to a corresponding command line parameter string.

2.3.8 Program Termination

A process executing a.out may terminate in two possible ways.

(1). Normal Termination: If the program executes successfully, main() eventually returns to crt0.o, which calls the library function exit(0) to terminate the process. The exit(value) function does some clean-up work first, such as flush stdout, close I/O streams, etc. Then it issues an _exit(value) **system call,** which causes the process to enter the OS kernel to terminate. A 0 exit value usually means normal termination. If desired, a process may call exit(value) directly without going back to crt0.o. Even more drastically, a process may issue an _exit(value) system call to terminate immediately without doing the clean-up work first. When a process terminates in kernel, it records the value in the _exit(value) system call as the exit status in the process structure, notifies its parent and becomes a ZOMBIE. The parent process can find the ZOMBIE child, get its pid and exit status by the system call

```
pid = wait(int *status);
```

which also releases the ZMOBIE child process structure as FREE, allowing it to be reused for another process.

(2). Abnormal Termination: While executing a.out the process may encounter an error condition, such as invalid address, illegal instruction, privilege violation, etc. which is recognized by the CPU as an **exception**. When a process encounters an exception, it is forced into the OS kernel by a trap. The kernel's trap handler converts the trap error type to a magic number, called a **SIGNAL**, and delivers the signal to the process, causing it to terminate. In this case, the exit status of the ZOMBIE process is the signal number, and we may say that the process has terminated abnormally. In addition to trap errors, signals may also originate from hardware or from other processes. For example, pressing the Control_C key generates a hardware interrupt, which sends the number 2 signal SIGINT to all processes on that terminal, causing them to terminate. Alternatively, a user may use the command

```
kill  -s signal_number pid    # signal_number = 1 to 31
```

to send a signal to a target process identified by pid. For most signal numbers, the default action of a process is to terminate. Signals and signal handling will be covered later in Chap. 6.

2.4 Function Call in C

Next, we consider the run-time behavior of a.out during execution. The run-time behavior of a program stems mainly from function calls. The following discussions apply to running C programs on 32-bit Intel x86 processors. On these machines, the C compiler generated code passes parameters on the stack in function calls. During execution, it uses a special CPU register (ebp) to point at the stack frame of the current executing function.

2.4.1 Run-Time Stack Usage in 32-Bit GCC

Consider the following C program, which consists of a main() function shown on the left-hand side, which calls a sub() function shown on the right-hand side.

```
-----------------------------------------------------------
  main()                        |     int sub(int x, int y)
  {                             |     {
    int a, b, c;                |         int u, v;
    a = 1; b = 2; c = 3;        |         u = 4; v = 5;
    c = sub(a, b);              |         return x+y+u+v;
    printf("c=%d\n", c);        |     }
  }                             |
-----------------------------------------------------------
```

(1) When executing a.out, a process image is created in memory, which looks (logically) like the diagram shown in Fig. 2.7, where Data includes both initialized data and bss.
(2) Every CPU has the following registers or equivalent, where the entries in parentheses denote registers of the x86 CPU:
PC (IP): point to next instruction to be executed by the CPU.
SP (SP): point to top of stack.
FP (BP): point to the stack frame of current active function.
Return Value Register (AX): register for function return value.
(3) In every C program, main() is called by the C startup code crt0.o. When crt0.o calls main(), it pushes the return address (the current PC register) onto stack and replaces PC with the entry address of main(), causing the CPU to enter main(). For convenience, we shall show the stack contents from left to right. When control enters main(), the stack contains the saved return PC on top, as shown in Fig. 2.8, in which XXX denotes the stack contents before crt0.o calls main(), and SP points to the saved return PC from where crt0.o calls main().
(4) Upon entry, the compiled code of every C function does the following:
. push FP onto stack # this saves the CPU's FP register on stack.
. let FP point at the saved FP # establish stack frame
. shift SP downward to allocate space for automatic local variables on stack
. the compiled code may shift SP farther down to allocate some temporary working space on the stack, denoted by temps.

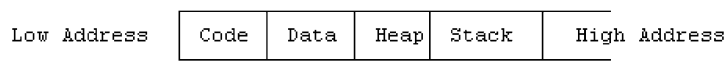


Fig. 2.7 Process execution image

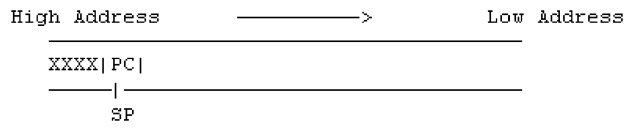


Fig. 2.8 Stack contents in function call

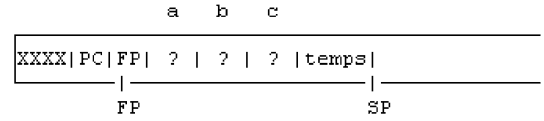


Fig. 2.9 Stack contents: allocate local variables

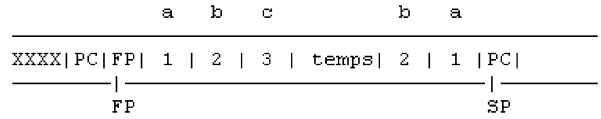


Fig. 2.10 Stack contents: passing parameters

For this example, there are 3 automatic local variables, int a, b, c, each of sizeof(int) = 4 bytes. After entering main(), the stack contents becomes as shown in Fig. 2.9, in which the spaces of a, b, c are allocated but their contents are yet undefined.

(5) Then the CPU starts to execute the code a=1; b=2; c=3; which put the values 1, 2, 3 into the memory locations of a, b, c, respectively. Assume that sizeof(int) is 4 bytes. The locations of a, b, c are at -4, -8, -12 bytes from where FP points at. These are expressed as -4(FP), -8(FP), -12(FP) in assembly code, where FP is the stack frame pointer. For example, in 32-bit Linux the assembly code for b=2 in C is

```
movl   $2, -8(%ebp)      # b=2 in C
```

where $2 means the value of 2 and %ebp is the ebp register.

(6) main() calls sub() by c = sub(a, b); The compiled code of the function call consists of
. Push parameters in reverse order, i.e. push values of b=2 and a=1 into stack.
. Call sub, which pushes the current PC onto stack and replaces PC with the entry address of sub, causing the CPU to enter sub().
When control first enters sub(), the stack contains a return address at the top, preceded by the parameters, a, b, of the caller, as shown in Fig. 2.10.

(7) Since sub() is written in C, it actions are exactly the same as that of main(), i.e. it
. Push FP and let FP point at the saved FP;
. Shift SP downward to allocate space for local variables u, v.
. The compiled code may shift SP farther down for some temp space on stack.
The stack contents becomes as shown in Fig. 2.11.

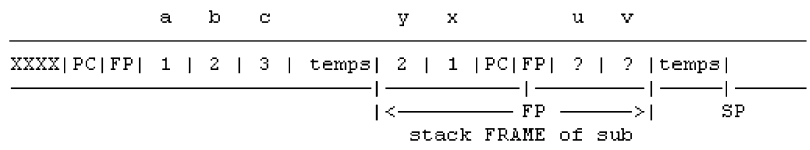


Fig. 2.11 Stack contents of called function

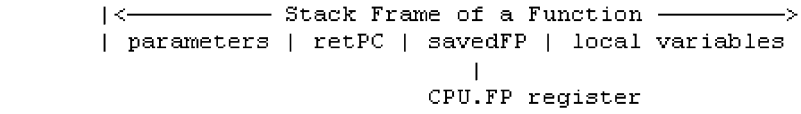


Fig. 2.12 Function stack frame

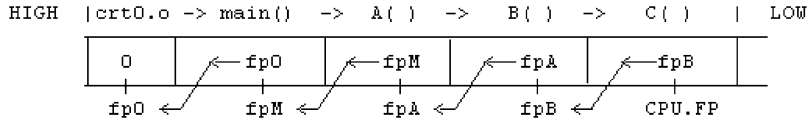


Fig. 2.13 Function call sequence

2.4.2 Stack Frames

While execution is inside a function, such as sub(), it can only access global variables, parameters passed in by the caller and local variables, but nothing else. Global and static local variables are in the combined Data section, which can be referenced by a fixed base register. Parameters and automatic locals have different copies on each invocation of the function. So the problem is: how to reference parameters and automatic locals? For this example, the parameters a, b, which correspond to the arguments x, y, are at 8(FP) and 12(FP). Similarly, the automatic local variables u, v are at -4(FP) and -8(FP). The stack area visible to a function, i.e. parameters and automatic locals, is called the **Stack Frame** of a function, like a frame of movie to a person. Thus, FP is called the **Stack Frame Pointer.** To a function, the stack frame looks like the following (Fig. 2.12).

From the above discussions, the reader should be able to deduce what would happen if we have a sequence of function calls, e.g.

```
crt0.o --> main() --> A(par_a) --> B(par_b) --> C(par_c)
```

For each function call, the stack would grow (toward low address) one more frame for the called function. The frame at the stack top is the stack frame of the current executing function, which is pointed by the CPU's frame pointer. The saved FP points (backward) to the frame of its caller, whose saved FP points back at the caller's caller, etc. Thus, the function call sequence is maintained in the stack as a link list, as shown in Fig. 2.13.

By convention, the CPU's FP = 0 when crt0.o is entered from the OS kernel. So the stack frame link list ends with a 0. When a function returns, its stack frame is deallocated and the stack shrinks back.

2.4.3 Return From Function Call

When sub() executes the C statement **return x+y+u+v**, it evaluates the expression and puts the resulting value in the return value register (AX). Then it deallocates the local variables by

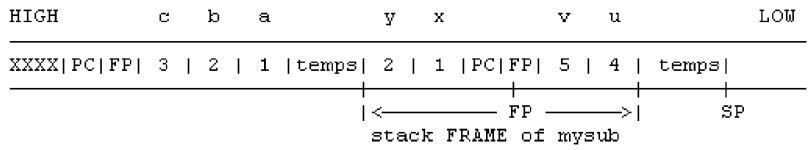


Fig. 2.14 Stack contents with reversed allocation scheme

.copy FP into SP; # SP now points to the saved FP in stack.
.pop stack into FP; # this restores FP, which now points to the caller's stack frame,
leaving the return PC on the stack top.
(On the x86 CPU, the above operations are equivalent to the **leave** instruction).
.Then, it executes the RET instruction, which pops the stack top into PC register, causing the CPU to execute from the saved return address of the caller.

(8) Upon return, the caller function catches the return value in the return register (AX). Then it cleans the parameters a, b, from the stack (by adding 8 to SP). This restores the stack to the original situation before the function call. Then it continues to execute the next instruction.

It is noted that some compilers, e.g. GCC Version 4, allocate automatic local variables in increasing address order. For instance, int a, b; implies (address of a) < (address of b). With this kind of allocation scheme, the stack contents may look like the following (Fig. 2.14).

In this case, automatic local variables are also allocated in "reverse order", which makes them consistent with the parameter order, but the concept and usage of stack frames remain the same.

2.4.4 Long Jump

In a sequence of function calls, such as

```
main() --> A() --> B()-->C();
```

when a called function finishes, it normally returns to the calling function, e.g. C() returns to B(), which returns to A(), etc. It is also possible to return directly to an earlier function in the calling sequence by a long jump. The following program demonstrates long jump in Unix/Linux.

```
/** longjump.c file: demonstrate long jump in Linux **/
#include <stdio.h>
#include <setjmp.h>
jmp_buf env;        // for saving longjmp environment

int main()
{
  int r, a=100;
  printf("call setjmp to save environment\n");
  if ((r=setjmp(env)) == 0){
     A();
     printf("normal return\n");
  }
```

```
   else
      printf("back to main() via long jump, r=%d a=%d\n", r, a);
}

int A()
{   printf("enter A()\n");
    B();
    printf("exit A()\n");
}

int B()
{
  printf("enter B()\n");
  printf("long jump? (y|n) ");
  if (getchar()=='y')
     longjmp(env, 1234);
  printf("exit B()\n");
}
```

In the longjump program, **setjmp()** saves the current execution environment in a jmp_buf structure and returns 0. The program proceeds to call A(), which calls B(). While in the function B(), if the user chooses not to return by long jump, the functions will show the normal return sequence. If the user chooses to return by **longjmp(env, value),** execution will return to the last saved environment with a nonzero value. In this case, it causes B() to return to main() directly, bypassing A(). The principle of long jump is very simple. When a function finishes, it returns by the (callerPC, callerFP) in the current stack frame, as shown in Fig. 2.15.

If we replace (callerPC, callerFP) with (savedPC, savedFP) of an earlier function in the calling sequence, execution would return to that function directly. In addition to the (savedPC, savedFP), **setjmp()** may also save CPU's general registers and the original SP, so that **longjmp()** can restore the complete environment of the returned function. Long jump can be used to abort a function in a calling sequence, causing execution to resume from a known environment saved earlier. Although rarely used in user mode programs, it is a common technique in systems programming. For example, it may be used in a signal catcher to bypass a user mode function that caused an exception or trap error. We shall demonstrate this technique later in Chap. 6 on signals and signal processing.

2.4.5 Run-Time Stack Usage in 64-Bit GCC

In 64-bit mode, the CPU registers are expanded to rax, rbx, rcx, rdx, rbp, rsp, rsi, rdi, r8 to r15, all 64-bit wide. The function call convention differs slightly from 32-bit mode. When calling a function, the first 6 parameters are passed in rdi, rsi, rdx, rcx, r8, r9, in that order. Any extra parameters are passed through the stack as they are in 32-bit mode. Upon entry, a called function first establishes the stack frame (using rbp) as usual. Then it may shift the stack pointer (rsp) downward for local variables and working spaces on the stack. The GCC compiler generated code may keep the stack pointer fixed,

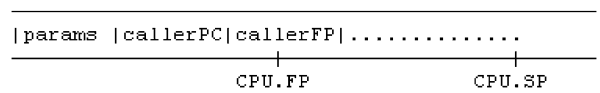


Fig. 2.15 Function return frame

with a default reserved **Red Zone** stack area of 128 bytes, while execution is inside a function, making it possible to access stack contents by using rsp as the base register. However, the GCC compiler generated code still uses the stack frame pointer rbp to access both parameters and locals. We illustrate the function call convention in 64-bit mode by an example.

Example: Function Call Convention in 64-Bit Mode

(1) The following t.c file contains a main() function in C, which defines 9 local int (32-bit) variables, a to i. It calls a sub() function with 8 int parameters.

```
/*********  t.c file ********/
#include <stdio.h>
int sub(int a, int b, int c, int d, int e, int f, int g, int h)
{
  int u, v, w;
  u = 9;
  v = 10;
  w = 11;
  return a+g+u+v; // use first and extra parameter, locals
}

int main()
{
  int a, b, c, d, e, f, g, h, i;
  a = 1;
  b = 2;
  c = 3;
  d = 4;
  e = 5;
  f = 6;
  g = 7;
  h = 8;
  i = sub(a,b,c,d,e,f,g,h);
}
```

(2) Under 64-bit Linux, compile t.c to generate a t.s file in 64-bit assembly by

```
gcc -S t.c  # generate t.s file
```

Then edit the t.s file to delete the nonessential lines generated by the compiler and add comments to explain the code actions. The following shows the simplified t.s file with added comment lines.

```
#------------ t.s file generated by 64-bit GCC compiler -------------
       .globl  sub
sub:              # int sub(int a,b,c,d,e,f,g,h)

# first 6 parameters a,  b,  c,  d,  e,  f are in registers
#                   rdi,rsi,rdx,rcx,r8d,r9d
# 2 extra parameters g,h are on stack.
```

```
# Upon entry, stack top contains g, h
#    ---------------------------------
#     ... ...| h | g | PC |    LOW address
#    ---------------|------------------
#                  rsp

# establish stack frame
       pushq  %rbp
       movq   %rsp, %rbp
# no need to shift rsp down because each function has a 128 bytes
# reserved stack area.
# rsp will be shifted down if function define more locals

# save first 6 parameters in registers on stack
       movl   %edi, -20(%rbp) # a
       movl   %esi, -24(%rbp) # b
       movl   %edx, -28(%rbp) # C
       movl   %ecx, -32(%rbp) # d
       movl   %r8d, -36(%rbp) # e
       movl   %r9d, -40(%rbp) # f

# access locals u, v, w at rbp -4 to -12
       movl   $9,   -4(%rbp)
       movl   $10,  -8(%rbp)
       movl   $11, -12(%rbp)

# compute x + g + u + v:
       movl   -20(%rbp), %edx   # saved a on stack
       movl   16(%rbp), %eax    # g at 16(rbp)
       addl   %eax, %edx
       movl   -4(%rbp), %eax    # u at -4(rbp)
       addl   %eax, %edx
       movl   -8(%rbp), %eax    # v at -8(rbp)
       addl   %edx, %eax

# did not shift rsp down, so just popQ to restore rbp
       popq   %rbp
       ret

#====== main function code in assembly ======
       .globl       main
main:
# establish stack frame
       pushq  %rbp
       movq   %rsp, %rbp

# shit rsp down 48 bytes for locals
       subq   $48, %rsp
```

```
# locals are at rbp -4 to -32
       movl   $1,  -4(%rbp)     # a=1
       movl   $2,  -8(%rbp)     # b=2
       movl   $3, -12(%rbp)     # c=3
       movl   $4, -16(%rbp)     # d=4
       movl   $5, -20(%rbp)     # e=5
       movl   $6, -24(%rbp)     # f=6
       movl   $7, -28(%rbp)     # g=7
       movl   $8, -32(%rbp)     # h=8

# call sub(a,b,c,d,e,f,g,h): first 6 parameters in registers
       movl   -24(%rbp), %r9d   # f in r9
       movl   -20(%rbp), %r8d   # e in r8
       movl   -16(%rbp), %ecx   # d in ecx
       movl   -12(%rbp), %edx   # c in edx
       movl    -8(%rbp), %esi   # b in esi
       movl    -4(%rbp), %eax   # a in eax but will be in edi
# push 2 extra parameters h,g on stack
       movl   -32(%rbp), %edi   # int h in edi
       pushq  %rdi              # pushQ rdi ; only low 32-bits = h
       movl   -28(%rbp), %edi   # int g in edi
       pushq  %rdi              # pushQ rdi ; low 32-bits = g

       movl   %eax, %edi        # parameter a in edi
       call   sub               # call sub(a,b,c,d,e,f,g,h)

       addq   $16, %rsp         # pop stack: h,g, 16 bytes
       movl   %eax, -36(%rbp)   # i = sub return value in eax

       movl   $0, %eax          # return 0 to crt0.o
       leave
       ret

# GCC compiler version 5.3.0
       .ident   "GCC: (GNU) 5.3.0"
```

2.5 Link C Program with Assembly Code

In systems programming, it is often necessary to access and control the hardware, such as CPU registers and I/O port locations, etc. In these situations, assembly code becomes necessary. It is therefore important to know how to link C programs with assembly code.

2.5.1 Programming in Assembly

(1) **C code to Assembly Code**

```
/************* a.c file ********************/
#include <stdio.h>

extern int B();

int A(int x, int y)
{
  int d, e, f;
  d = 4; e = 5; f = 6;
  f = B(d,e);
}

======= compile a.c file into 32-bit assembly code ======
cc -m32 -S a.c ===> a.s file

=====================================================
        .text
        .globl A
A:
         pushl    %ebp
         movl     %esp, %ebp

         subl     $24, %esp
         movl     $4, -12(%ebp)    # d=4
         movl     $5,  -8(%ebp)    # e=5
         movl     $6,  -4(%ebp)    # f=6

         subl     $8, %esp
         pushl     -8(%ebp)        # push e
         pushl    -12(%ebp)        # push d
         call     B

         addl     $16, %esp        # clean stack
         movl     %eax, -4(%ebp)   # f=return value in AX

         leave
         ret

=====================================================
```

Explanations of the Assembly Code
The assembly code generated by GCC consists of three parts:

(1). Entry: also called the **prolog**, which establishes stack frame, allocates local variables and working space on stack

(2). Function body, which performs the function task with return value in AX register
(3). Exit: also called the **epilog**, which deallocates stack space and return to caller

The GCC generated assembly code are explained below, along with the stack contents

```
A:                                  # A() start code location
(1). Entry Code:
        pushl   %ebp
        movl    %esp, %ebp          # establish stack frame
```

The entry code first saves FP (%bp) on stack and let FP point at the saved FP of the caller. The stack contents become

```
        SP
---------|--------------------------- LOW address
 xxx |PC|FP|
---------|---------------------------
        FP

        subl    $24, %esp
```

Then it shift SP downward 24 bytes to allocate space for locals variables and working area.

```
(2). Function Body Code:
        movl    $4, -20(%ebp)  // d=4
        movl    $5, -16(%ebp)  // e=5
        movl    $6, -12(%ebp)  // f=6
```

While inside a function, FP points at a fixed location and acts as a base register for accessing local variables, as well as parameters. As can be seen, the 3 locals d, e, f, each 4 bytes long, are at the byte offsets -20, -16, -12 from FP. After assigning values to the local variables, the stack contents become

```
        SP  ---   -24  ----> SP
        |                    |
---------- -4  -8 -12 -16 -20 -24|------------------ LOW address
 xxx |PC|FP|? |? | 6 | 5 | 4 |?  |
---------|---------f---e---d----|------------------
        FP

# call B(d,e): push parameters d, e in reverse order:

        subl    $8, %esp        # create 8 bytes TEMP slots on stack
        pushl   -16(%ebp)       # push e
        pushl   -20(%ebp)       # push d

                                                        SP
 -------------|-4 -8 -12 -16 -20 -24|- TEMP -|-------|--- LOW address
 xxx |retPC|FP|? |? | 6 | 5 | 4 |?  | ??| ?? | e | d |
 ----------|---------f---e---d----|--------------------
           FP
```

```
        call    B               # B() will grow stack to the RIGHT

# when B() returns:
        addl    $16, %esp       # clean stack
        movl    %eax, -4(%ebp)  # f = return value in AX

(3). Exit Code:
#      leave
         movl   %ebp, %esp      # SAME as leave
         popl   %ebp

         ret                    # pop retPC on stack top into PC
```

2.5.2 Implement Functions in Assembly

Example 1: Get CPU registers. Since these functions are simple, they do not need to establish and deallocate stack frames.

```
#============== s.s file ===============
         .global get_esp, get_ebp
get_esp:
         movl    %esp, %eax
         ret
get_ebp:
         movl    %ebp, %eax
         ret
#=====================================

int main()
{
   int ebp, esp;
   ebp = get_ebp();
   esp = get_esp();
   printf("ebp=%8x   esp=%8x\n", ebp, esp);
}
```

Example 2: Assume int mysum(int x, int y) returns the sum of x and y. Write mysum() function in ASSEMBLY. Since the function must use its parameters to compute the sum, we show the entry, function body and exit parts of the function code.

```
# ============ mysum.s file ============================
        .text                   # Code section
        .global mysum, printf    # globals: export mysum, import printf
mysum:
```

```
# (1) Entry:(establish stack frame)

        pushl %ebp
        movl  %esp, %ebp

# Caller has pushed y, x on stack, which looks like the following
#            12   8    4     0
#-----------------------------------------------------
#          |  y | x |retPC| ebp|
#----------------------------|----------------------
#                           ebp

# (2): Function Body Code of mysum: compute x+y in AX register
        movl  8(%ebp), %eax    # AX = x
        addl 12(%ebp), %eax    # AX += y

# (3) Exit Code: (deallocate stack space and return)
        movl  %ebp, %esp
        pop   %ebp
        ret
# =========== end of mysum.s file ==========================

int main()  # driver program to test mysum() function
{
  int a,b,c;
  a = 123; b = 456;
  c = mysum(a, b);
  printf("c=%d\n", c);   // c should be 579
}
```

2.5.3 Call C functions from Assembly

Example 3: Access global variables and call printf()

```
int a, b;
int main()
{
   a = 100; b = 200;
   sub();
}

#========== Assembly Code file ================
        .text
        .global sub, a, b, printf
sub:
        pushl   %ebp
        movl    %esp, %ebp
```

```
        pushl   b
        pushl   a
        pushl   $fmt          # push VALUE (address) of fmt
        call    printf        # printf(fmt, a, b);
        addl    $12, %esp

        movl    %ebp, %esp
        popl    %ebp
        ret

        .data
fmt:    .asciz   "a=%d  b=%d\n"
#====================================
```

2.6 Link Library

A link library contains precompiled object code. During linking, the linker uses the link library to complete the linking process. In Linux, there are two kinds of link libraries; **static link library** for static linking, and **dynamic link library** for dynamic linking. In this section, we show how to create and use link libraries in Linux.

Assume that we have a function

```
// musum.c file
int mysum(int x, int y){ return x + y; }
```

We would like to create a link library containing the object code of the mysum() function, which can be called from different C programs, e.g.

```
// t.c file
int main()
{
   int sum = mysum(123,456);
}
```

2.6.1 Static Link Library

The following steps show how to create and use a static link library.

```
(1). gcc  -c  mysum.c                # compile mysum.c into mysum.o
(2). ar   rcs   libmylib.a   mysum.o  # create static link library with member
                                       mysum.o
(3). gcc  -static  t.c  -L. -lmylib  # static compile-link t.c with libmylib.a
                                       as link library
(4). a.out                           # run a.out as usual
```

In the compile-link step (4), -L. specifies the library path (current directory), and -l specifies the library. Note that the library (mylib) is specified without the prefex lib, as well as the suffix .a

2.6.2 Dynamic Link Library

The following steps show how to create and use a dynamic link library.

```
(1). gcc  -c  -fPIC  mysum.c              # compile to Position Independent
                                            Code mysum.o
(2). gcc -shared  -o libmylib.so  mysum.o  # create shared libmylib.so with
                                            mysum.o
(3). gcc  t.c  -L.  -lmylib               # generate a.out using shared library
                                            libmylib.so
(4). export LD_LIBRARY_PATH=./            # to run a.out, must export
                                            LD_LIBRARY=./
(5). a.out                                #  run a.out. ld will load libmylib.so
```

In both cases, if the library is not in the current directory, simply change the –L. option and set the LD_LIBRARY_PATH to point to the directory containing the library. Alternatively, the user may also place the library in a standard lib directory, e.g. /lib or /usr/lib and run ldconfig to configure the dynamic link library path. The reader may consult Linux ldconfig (man 8) for details.

2.7 Makefile

So far, we have used individual gcc commands to compile-link the source files of C programs. For convenience, we may also use a **sh script** which includes all the commands. These schemes have a major drawback. If we only change a few of the source files, the sh commands or script would still compile all the source files, including those that are not modified, which is unnecessary and time-consuming. A better way is to use the Unix/Linux **make** facility (GNU make 2008). make is a program, which reads a **makefile**, or **Makefile** in that order, to do the compile-link automatically and selectively. This section covers the basics of makefiles and shows their usage by examples.

2.7.1 Makefile Format

A make file consists of a set of **targets**, **dependencies** and **rules.** A **target** is usually a file to be created or updated, but it may also be a directive to, or a label to be referenced by, the make program. A target depends on a set of source files, object files or even other targets**,** which are described in a **Dependency List**. **Rules** are the necessary commands to build the target by using the Dependency List. Figure 2.16 shows the format of a makefile.

2.7.2 The make Program

When the **make** program reads a makefile, it determines which targets to build by comparing the timestamps of source files in the Dependency List. If any dependency has a newer timestamp since last

Fig. 2.16 Makefile format

Target	Dependency List
target:	file1 file2 … fileN
	Rules
<tab>	command1
<tab>	command2
<tab>	other command

build, **make** will execute the rule associated with the target**.** Assume that we have a C program consisting of three source files:

```
(1). type.h file:                  // header file
     int mysum(int x, int y)       // types, constants, etc

(2). mysum.c file:                 // function in C
     #include <stdio.h>
     #incldue "type.h"
     int mysum(int x, int y)
     {
         return x+y;
     }

(3). t.c file:                     // main() in C
     #include <stdio.h>
     #include "type.h"
     int main()
     {
         int sum = mysum(123,456);
         printf("sum = %d\n", sum);
     }
```

Normally, we would use the sh command

```
gcc -o myt main.c mysum.c
```

to generate a binary executable named **myt**. In the following, we shall demonstrate compile-link of C programs by using makefiles.

2.7.3 Makefile Examples

Makefile Example 1

(1). Create a makefile named mk1 containing:

```
myt: type.h t.c mysum.c          # target: dependency list
       gcc -o myt t.c mysum.c    # rule: line MUST begin with a TAB
```

The resulting executable file name, myt in this example, usually matches that of the target name. This allows make to decide whether or not to build the target again later by comparing its timestamp against those in the dependency list.

(2). Run make using mk1 as the makefile: **make** normally uses the default makefile or Makefile, whichever is present in the current directory. It can be directed to use a different makefile by the –f flag, as in

```
make –f mk1
```

make will build the target file myt and show the command execution as

```
gcc –o myt  t.c  mysum.c
```

(3). Run the make command again. It will show the message

```
make: 'myt' is up to date
```

In this case, make does not build the target again since none of the files has changed since last build.

(4). On the other hand, make will execute the rule command again if any of the files in the dependency list has changed. A simple way to modify a file is by the **touch** command, which changes the timestamp of the file. So if we enter the sh commands

```
touch type.h     // or touch *.h, touch *.c, etc.
make –f mk1
```

make will recompile-link the source files to generate a new myt file

(5). If we delete some of the file names from the dependency list, make will not execute the rule command even if such files are changed. The reader may try this to verify it.

As can be seen, mk1 is a very simple makfile, which is not much different than sh commands. But we can refine makefiles to make them more flexible and general.

Makefile Example 2: Macros in Makefile

(1). Create a makefile named mk2 containing:

```
CC = gcc                  # define CC as gcc
CFLAGS = -Wall            # define CLAGS as flags to gcc
OBJS = t.o mysum.o        # define Object code files
INCLUDE = -Ipath          # define path as an INCLUDE directory

myt: type.h $(OBJS)       # target: dependency: type.h and .o files
        $(CC) $(CFLAGS) –o t $(OBJS) $(INCLUDE)
```

In a makefile, macro defined symbols are replaced with their values by $(symbol), e.g. $(CC) is replaced with gcc, $(CFLAGS) is replaced with –Wall, etc. For each .o file in the dependency list,

make will compile the corresponding .c file into .o file first. However, this works only for .c files. Since all the .c files depend on .h files, we have to explicitly include type.h (or any other .h files) in the dependency list also. Alternatively, we may define additional targets to specify the dependency of .o files on .h files, as in

```
t.o:      t.c type.h          # t.o depend on t.c and type.h
       gcc -c t.c
mysum.o:  mysum.c type.h      # mysum.o depend type.h
       gcc -c mysum.c
```

If we add the above targets to a makefile, any changes in either .c files or type.h will trigger make to recompile the .c files. This works fine if the number of .c files is small. It can be very tedious if the number of .c files is large. So there are better ways to include .h files in the dependency list, which will be shown later.

(3). Run make using mk2 as the makefile:

```
make -f mk2
```

(4). Run the resulting binary executable myt as before.

The simple makefiles of Examples 1 and 2 are sufficient for compile-link most small C programs. The following shows some additional features and capabilities of makefiles.

Makefile Example 3: Make Target by Name
When make runs on a makefile, it normally tries to build the **first target** in the makefile. The behavior of make can be changed by specifying a target name, which causes make to build the specific named target. As an example, consider the makefile named mk3, in which the new features are highlighted in **bold face** letters.

```
# ------------------  mk3 file --------------------
CC = gcc                 # define CC as gcc
CFLAGS = -Wall           # define CLAGS as flags to gcc
OBJS = t.o mysum.o       # define Object code files
INCLUDE = -Ipath         # define path as an INCLUDE directory

all: myt install         # build all listed targets: myt, install

myt: t.o mysum.o         # target: dependency list of .o files
       $(CC) $(CFLAGS) -o myt $(OBJS) $(INCLUDE)

t.o:    t.c  type.h      #  t.o depend on t.c and type.h
       gcc -c t.c
mysum.o: mysum.c type.h #  mysum.o depend mysum.c and type.h
      gcc -c mysum.c

install: myt             # depend on myt: make will build myt first
       echo install myt to /usr/local/bin
       sudo mv myt /usr/local/bin/  # install myt to /usr/local/bin/
```

```
run:    install          # depend on install, which depend on myt
       echo run executable image myt
       myt || /bin/true  # no make error 10 if main() return non-zero

clean:
      rm -f *.o 2> /dev/null            # rm all *.o files
      sudo rm -f /usr/local/bin/myt     # rm myt
```

The reader may test the mk3 file by entering the following **make** commands:

```
(1). make [all] -f mk3          # build all targets: myt and install
(2). make install -f mk3        # build target myt and install myt
(3). make run -f mk3            # run /usr/local/bin/myt
(4). make clean -f mk3          # remove all listed files
```

Makefile Variables: Makefiles support variables. In a makefile, **%** is a wildcard variable similar to * in sh. A makefile may also contain **automatic variables**, which are set by make after a rule is matched. They provide access to elements from the target and dependency lists so that the user does not have to explicitly specify any filenames. They are very useful for defining general pattern rules. The following lists some of the automatic variables of make.

$@ : name of current target.
$< : name of first dependency
$^ : names of all dependencies
$* : name of current dependency without extension
$? : list of dependencies changed more recently than current target.

In addition, make also supports **suffix rules,** which are not targets but directives to the make program. We illustrate make variables and suffix rules by an example.

In a C program, .c files usually depend on all .h files. If any of the .h files is changed, all .c files must be re-compiled again. To ensure this, we may define a dependency list containing all the .h files and specify a target in a makefile as

```
DEPS = type.h          # list ALL needed .h files
%.o: %.c $(DEPS)       # for all .o files: if its .c or .h file changed
       $(CC) -c -o $@  # compile corresponding .c file again
```

In the above target, %.o stands for all .o files and $@ is set to the current target name, i.e. the current .o file name. This avoids defining separate targets for individual .o files.

Makefile Example 4: Use make variables and suffix rules

```
# ---------- mk4 file -------------
CC = gcc
CFLAGS = -I.
OBJS = t.o mysum.o
AS = as     # assume we have .s files in assembly also
DEPS = type.h              # list all .h files in DEPS
```

```
.s.o: # for each fname.o, assemble fname.s into fname.o
        $(AS) -o $< -o $@  # -o $@ REQUIRED for .s files

.c.o: # for each fname.o, compile fname.c into fname.o
        $(CC) -c $< -o $@  # -o $@ optional for .c files

%.o:  %.c $(DEPS) # for all .o files: if its .c or .h file changed
        $(CC) -c -o $@ $<  # compile corresponding .c file again

myt: $(OBJS)
        $(CC) $(CFLAGS) -o $@ $^
```

In the makefile mk4, the lines .s.o: and .c.o: are not targets but directives to the make program by the **suffix rule.** These rules specify that, for each .o file, there should be a corresponding .s or .c file to build if their timestamps differ, i.e. if the .s or .c file has changed. In all the target rules, $@ means the current target, $< means the first file in the dependency list and $^ means all files in the dependency list. For example, in the rule of the myt target, -o $@ specifies that the output file name is the current target, which is myt. $^ means it includes all the files in the dependency list, i.e. both t.o and mysum.o. If we change $^ to $< and touch all the .c files, make would generate an "undefined reference to mysum" error. This is because $< specifies only the first file (t.o) in the dependency list, make would only recompile t.c but not mysum.c, resulting a linking error due to missing mysum.o file. As can be seen from the example, we may use make variables to write very general and compact makefiles. The downside is that such makefiles are rather hard to understand, especially for beginning programmers.

Makfiles in Subdirectories

A large C programming project usually consists of tens or hundreds of source files. For ease of maintenance, the source files are usually organized into different levels of directories, each with its own makefile. It's fairly easy to let make go into a subdirectory to execute the local makefile in that directory by the command

```
(cd DIR; $(MAKE))   OR   cd DIR && $(MAKE)
```

After executing the local makefile in a subdirectory, control returns to the current directory form where make continues. We illustrate this advanced capability of make by a real example.

Makefile Example 5: PMTX System Makefiles

PMTX (Wang 2015) is a Unix-like operating system designed for the Intel x86 architecture in 32-bit protect mode. It uses 32-bit GCC assembler, compiler and linker to generate the PMTX kernel image. The source files of PMTX are organized in three subdirectories:

Kernel : PMTX kernel files; a few GCC assembly files, mostly in C
Fs : file system source files; all in C
Driver : device driver source files; all in C

The compile-link steps are specified by Makefiles in different directories. The top level makefile in the PMTX source directory is very simple. It first cleans up the directories. Then it goes into the Kernel subdirectory to execute a Makefile in the Kernel directory. The Kernel Makefile first generates .o files

for both .s and .c files in Kernel. Then it directs make to go into the Driver and Fs subdirectories to generate .o file by executing their local Makfiles. Finally, it links all the .o files to a kernel image file. The following shows the various Makefiles of the PMTX system.

```
#------------- PMTX Top level Makefile -------------
all: pmtx_kernel

pmtx_kernel:
       make clean
       cd Kernel && $(MAKE)

clean: # rm mtx_kerenl, *.o file in all directories

#------------- PMTX Kernel Makefile ----------------
AS = as –Iinclude
CC = gcc
LD = ld
CPP = gcc -E -nostdinc
CFLAGS = -W -nostdlib -Wno-long-long -I include -fomit-frame-pointer

KERNEL_OBJS = entry.o init.o t.o ts.o traps.o trapc.o queue.o \
fork.o exec.o wait.o io.o syscall.o loader.o pipe.o mes.o signal.o \
threads.o sbrk.o mtxlib.o

K_ADDR=0x80100000      # kernel start virtual address

all: kernel

.s.o:   # build each .o if its .s file has changed
         ${AS} -a $< -o $*.o > $*.map

pmtx_kernel: $(KERNEL_OBJS)      # kernel target: depend on all OBJs
         cd ../Driver && $(MAKE) # cd to Driver, run local Makefile
         cd ../Fs && $(MAKE)     # cd to Fs/,    run local Makefile

# link all .o files with entry=pm_entry, start VA=0x80100000
         ${LD} --oformat binary -Map k.map -N -e pm_entry \
              -Ttext ${K_ADDR} -o $@ \
               ${KERNEL_OBJS} ../DRIVER/*.o ../FS/*.o
clean:
         rm -f *.map *.o
         rm –f ../DRIVER.*.map ../DRIVER/*.o
         rm –f ../FS/*.map ../FS/*.o
```

The PMTX kernel makfile first generates .o files from all .s (assembly) files. Then it generates other .o files from .c files by the dependency lists in **KERNEL_OBJ.** Then it goes into **Driver** and **Fs** directories to execute the local makefiles, which generate .o files in these directories. Finally, it links all the .o files to generate the pmtx_kernel image file, which is the PMTX OS kernel. In contrast, since

all files in **Fs** and **Driver** are in C, their Makefiles only compile .c files to .o files, so there are no .s or ld related targets and rules.

```
#------------- PMTX Driver Makefile ----------------
CC=gcc
CPP=gcc -E -nostdinc
CFLAGS=-W -nostdlib -Wno-long-long -I include -fomit-frame-pointer

DRIVER_OBJS = timer.o pv.o vid.o kbd.o fd.o hd.o serial.o pr.o atapi.o
driverobj: ${DRIVER_OBJS}

#------------- PMTX Fs Makefile ------------------
CC=gcc
CPP=gcc -E -nostdinc
CFLAGS=-W -nostdlib -Wno-long-long -I include -fomit-frame-pointer

FS_OBJS = fs.o buffer.o util.o mount_root.o alloc_dealloc.o \
          mkdir_creat.o cd_pwd.o rmdir.o link_unlink.o stat.o touch.o \
          open_close.o read.o write.o dev.o mount_umount.o
fsobj:   ${FS_OBJS}
#--------------- End of Makefile -------------------
```

2.8 The GDB Debugger

The GNU Debugger **(GDB)** (Debugging with GDB 2002; GDB 2017) is an interactive debugger, which can debug programs written in C, C++ and several other languages. In Linux, the command **man gdb** displays the manual pages of gdb, which provides a brief description of how to use GDB. The reader may find more detailed information on GDB in the listed references. In this section, we shall cover GDB basics and show how to use GDB to debug C programs in the Integrated Development Environment (IDE) of EMACS under X-windows, which is available in all Linux systems. Since the Graphic User Interface (GUI) part of different Linux distributions may differ, the following discussions are specific to Ubuntu Linux Version 15.10 or later, but it should also be applicable to other Linux distributions, e.g. Slackware Linux 14.2, etc.

2.8.1 Use GDB in Emacs IDE

1. **Source Code:** Under X-window, open a pseudo-terminal. Use EMACS to create a Makefile, as shown below.

Makefile:

```
t: t.c
       gcc -g -o t t.c
```

Then use EMACS to edit a C source file. Since the objective here is to show GDB usage, we shall use a very simple C program.

```
/******** Source file: t.c Code ********/
#include <stdio.h>
int sub();
int g, h;   // globals

int main()
{
  int a, b, c;
  printf("enter main\n");
  a = 1;
  b = 2;
  c = 3;
  g = 123;
  h = 456;
  c = sub(a, b);
  printf("c = %d\n", c);
  printf("main exit\n");
}

int sub(int x, int y)
{
  int u,v;
  printf("enter sub\n");
  u = 4;
  v = 5;
  printf("sub return\n");
  return x+y+u+v+g+h;
}
```

2. **Compile Source Code:** When EMACS is running, it displays a menu and a tool bar at the top of the edit window (Fig. 2.17).

Each menu can be opened to display a table of submenus. Open EMACS **Tools** menu and select **Compile.** EMACS will show a prompt line at the bottom of the edit window

```
make -k
```

and waits for user response. EMACS normally compile-link the source code by a makefile. If the reader already has a makefile in the same directory as shown above, press the Enter key to let EMACS continue. In instead of a makefile, the reader may also enter the command line manually.

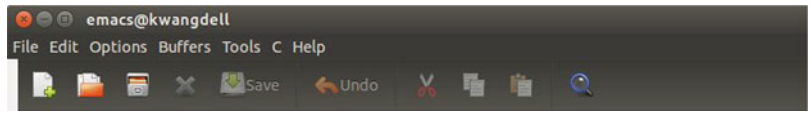


Fig. 2.17 EMACS menu and tool bar

Fig. 2.18 GDB menu and tool bar

```
gcc -g  -o  t  t.c
```

In order to generate a binary executable for GDB to debug, the **–g** flag is required. With the –g flag, the GCC compiler-linker will build a symbol table in the binary executable file for GDB to access variables and functions during execution. Without the **–g** flag, the resulting executable file can not be debugged by GDB. After compilation finishes, EMACS will show the compile results, including warning or error messages, if any, in a separate window below the source code window.

3. **Start up GDB:** Open EMACS Tools menu and select Debugger.

   EMACS will show a prompt line at the bottom of the edit window and wait for user response.

```
gdb -i=mi  t
```

Press Enter to start up the GDB debugger. GDB will run in the upper window and display a menu and a tool bar at the top of the EMACS edit window, as shown in Fig. 2.18.

The user may now enter GDB commands to debug the program. For example, to set break points, enter the GDB commands

```
b main       # set break point at main
b sub        # set break point at sub
b 10         # set break point at line 10 in program
```

When the user enters the Run (r) command (or choose **Run** in the tool bar), GDB will display the program code in the same GDB window. Other frames/windows can be activated through the submenu **GDB-Frames** or **GDB-Windows.** The following steps demonstrate the debugging process by using both commands and tool bar in the **multi-windows** layout of GDB.

4. **GDB in Multi-Windows:** From the GDB menu, choose **Gud => GDB-MI => Display Other Windows,** where => means follow a submenu**.** GDB will display GDB buffers in different windows, as shown in Fig. 2.19.

   Figure 2.19 shows six (6) GDB windows, each displays a specific GDB buffer.

Gud-t: GDB buffer for user commands and GDB messages
t.c: Program source code to show progress of execution
Stack frames: show stack frames of function calling sequence
Local Registers: show local variables in current executing function
Input/output: for program I/O
Breakpoints: display current break points settings

It also shows some of the commonly used GDB commands in a tool bar, e.g. **Run, Continue, Next line, Step line**, which allows the user to choose an action instead of entering commands.

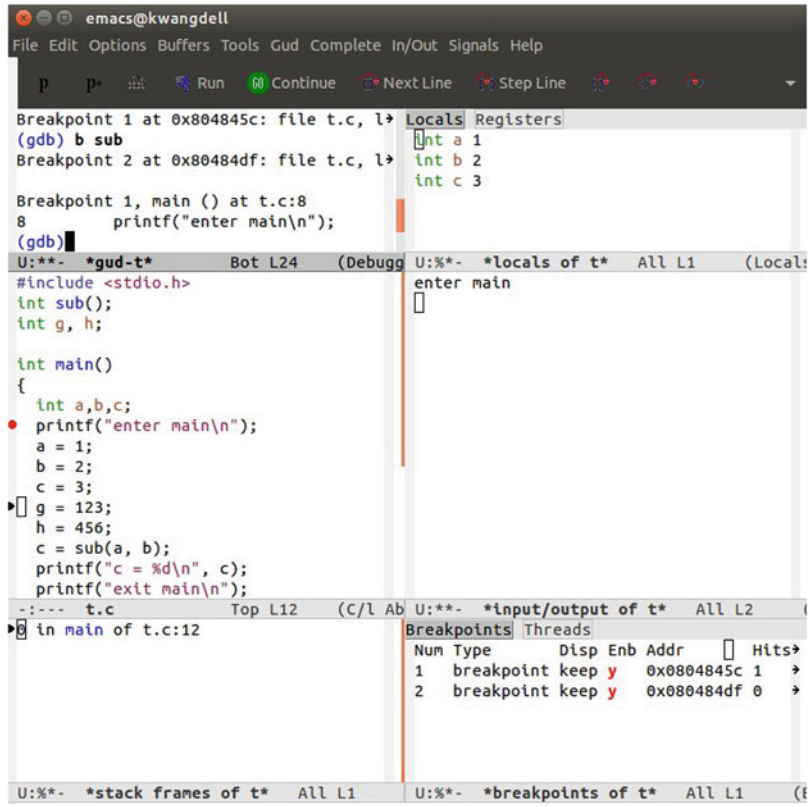


Fig. 2.19 Multi-windows of GDB

While the program is executing, GDB shows the execution progress by a **dark triangular mark**, which points to the next line of program code to be executed. Execution will stop at each break point, providing a break for the user to interact with GDB. Since we have set main as a break point, execution will stop at main when GDB starts to run. While execution stops at a break point, the user may interact with GDB by entering commands in the GDB window, such as set/clear break points, display/change variables, etc. Then, enter the Continue (c) command or choose **Continue** in the tool bar to continue program execution. Enter Next (n) command or choose **Next line** or **Step line** in the GDB tool bar to execute in single line mode.

Figure 2.19 shows that execution in main() has already completed executing several lines of the program code, and the next line mark is at the statement

$$g = 123;$$

At the moment, the local variables a, b, c are already assigned values, which are shown in the **Locals registers** windows as a=1, b=2, c=3. Global variables are not shown in any window, but the user may enter **Print** (p) commands

```
p g
p h
```

to print the global variables g and h, both of which should still be 0 since they are not assigned any values yet.

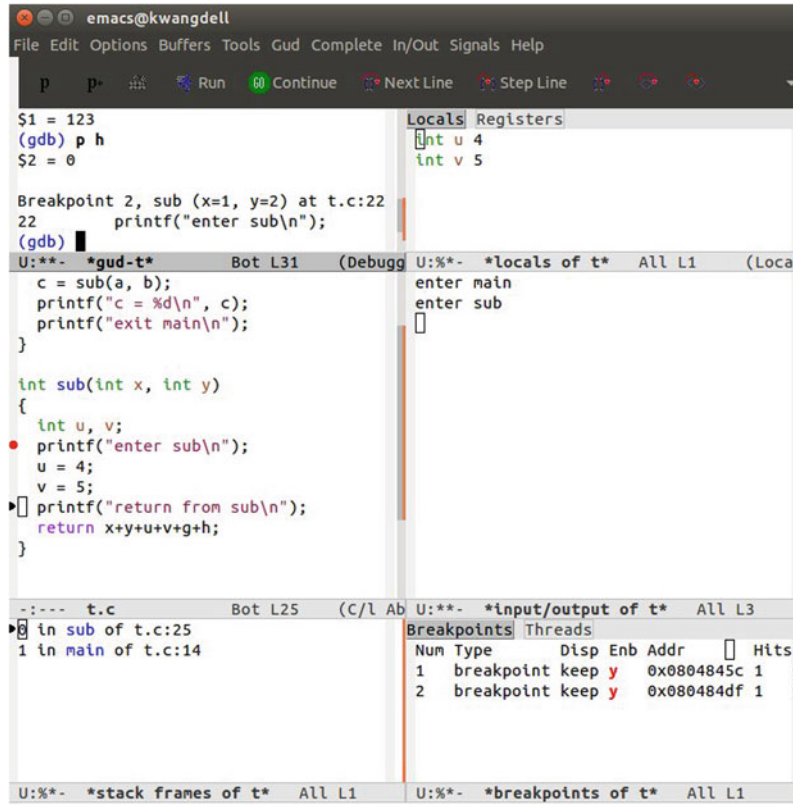


Fig. 2.20 Multi-windows of GDB

Figure 2.19 also shows the following GDB windows:

. **input/output** window shows the outputs of printf statements of main().
. **stack frames** window shows execution is now inside the main() function.
. **breakpoints** window shows the current breakpoints settings, etc.

This multi-windows layout provides the user with a complete set of information about the status of the executing program.

The user may enter Continue or choose **Continue** in the tool bar to continue the execution. When control reaches the sub() function, it will stop again at the break point. Figure 2.20 shows that the program execution is now inside sub() and the execution already passed the statements before

```
printf("return from sub\n");
```

At this moment, the **Locals Registers** window shows the local variables of sub() as u=4 and v=5. The **input/output** window shows the print results of both main() and sub(). The **Stack frame**s window shows sub() is the top frame and main() is the next frame, which is consistent with the function calling sequence.

(5). Additional GDB Commands: At each break point or while executing in single line mode, the user may enter GDB commands either manually, by the GDB tool bar or by choosing submenu items in the **Gud** menu, which includes all the commands in the GDB tool bar. The following lists some additional GDB commands and their meanings.

Clear Break Points:
clear line# : clear bp at line#
clear name : clear bp at function name

Change Variable Values
set var a=100 : set variable a to 100
set var b=200 : set b to 200, etc.

Watch Variable Changes:
watch c : watch for changes in variable c; whenever c changes, it will display its old value and new value.

Back trace (bt):
bt stackFrame# to back trace stack frames

2.8.2 Advices on Using Debugging Tools

GDB is a powerful debugger, which is fairly easy to use. However, the reader should keep in mind that all debugging tools can only offer limited help. In some cases, even a powerful debugger like the GDB is of little use. The best approach to program development is to design the program's algorithm carefully and then write program code in accordance with the algorithm. Many beginning programmers tend to write program code without any planning, just hoping their program would work, which most likely would not. When their program fails to work or does not produce the right results, they would immediately turn to a debugger, trying to trace the program executions to find out the problem. Relying too much on debugging tools is often counter-productive as it may waste more time than necessary. In the following, we shall point out some common programming errors and show how to avoid them in C programs.

2.8.3 Common Errors in C programs

A program in execution may encounter many types of run-time errors, such as illegal instruction, privilege violation, divide by zero, invalid address, etc. Such errors are recognized by the CPU as **exceptions**, which trap the process to the operating system kernel. If the user has not made any provision to handle such errors, the process will terminate by a signal number, which indicates the cause of the exception. If the program is written in C, which is executed by a process in user mode, exceptions such as illegal instruction and privilege violation should never occur. In system programming, programs seldom use divide operations, so divide by zero exceptions are also rare. The predominate type of run-time errors are due to invalid addresses, which cause memory access exceptions, resulting in the dreadful and familiar message of segmentation fault. In the following, we list some of the most probable causes in C programs that lead to memory access exceptions at run-time.

(1). Uninitialized pointers or pointers with wrong values: Consider the following code segments, with line numbers for ease of reference.

```
1. int *p;          // global, initial value = 0
   int main()
   {
2.     int *q;      // q is local on stack, can be any value
3.     *p = 1;      // dereference a NULL pointer
4.     *q = 2;      // dereference a pointer with unknown value
   }
```

Line 1 defines a global integer pointer p, which is in the BSS section of the run-time image with an initial value 0. So it's a NULL pointer. Line 3 tries to dereference a NULL pointer, which will cause a segmentation fault.

Line 2 defines a local integer pointer q, which is on the stack, so it can be any value. Line 4 tries to dereference the pointer q, which points at an unknown memory location. If the location is outside of the program's writable memory area, it will cause a segmentation fault due to memory access violation. If the location is within the program's writable memory area, it may not cause an immediate error but it may lead to other errors later due to corrupted data or stack contents. The latter kind of run-time error is extremely difficult to diagnose because the errors may have propagated through the program execution. The following shows the correct ways of using these pointers. Modify the above code segment as shown below.

```
int  x, *p;              // or  int *p = &x;
int main()
{
    int *q;
    p = &x;              // let p point at x
    *p = 1;
    q = (int *)malloc(sizeof(int);  // q point at allocate memory
    *q = 2;
 }
```

The principle is very simple. When using any pointer, the programmer must ensure the pointer is not NULL or has been set to point to a valid memory address.

(2). Array index out of bounds: In C programs, each array is defined with a finite number of N elements. The index of the array must be in the range of [0, N-1]. If the array index exceeds the range at run-time, it may cause invalid memory access, which either corrupt the program data area or result in a segmentation fault. We illustrate this by an example. Consider the following code segment.

```
     #define N 10
1.   int a[N], i; //  An array of N elements, followed by int i
     int main()
     {
2.      for (i=0; i<N; i++)      // index i in range
           a[i] = i+1;           // set a[ ] values = 1 to N
3.      a[N] = 123456789;        // set a[N] to a LARGE value
4.      printf("i = %d\n", i);   // print current i value
5.      printf("%d\n", a[i]));   // segmentation fault !
     }
```

Line 1 defines an array of N elements, which is followed by the index variable i. Line 2 represents the proper usage of the array index, which is within the bounds [0, N-1]. Line 3.sets a[N] to a large value, which actually changes the variable i because a[N] and i are in the same memory location. Line 4 prints the current value of i, which is no longer N but the large value. Line 5 will most likely to cause a segmentation fault because a[123456789] tries to access a memory location outside of the program's data area.

(3). Improper use of string pointers and char arrays: Many string operation functions in the C library are defined with char * parameters. As a specific example, consider the strcpy() function, which is defined as

```
char * strcpy(char *dest, char *src)
```

It copies a string from src to dest. The Linux man page on strcpy() clearly specifies that the dest string must be large enough to receive the copy. Many programmers, including some 'experienced' graduate students, often overlook the specification and try to use strcpy() as follows.

```
char *s;  // s is a char pointer
strcpy(s, " this is a string");
```

The code segment is wrong because s is not pointing at any memory location with enough space to receive the src string. If s is global, it is a NULL pointer. In this case, strcpy() will cause a segmentation fault immediately. If s is local, it may point to an arbitrary memory location. In this case, strcpy() may not cause an immediate error but it may lead to other errors later due to corrupted memory contents. Such errors are very subtle and difficult to diagnose even with a debugger such as GDB. The correct way of using strcpy() is to ensure dest is NOT just a string pointer but a real memory area with enough space to receive the copied string, as in

```
char s[128];  // s is a char array
strcpy(s, " this is a string");
```

Although the same s variable in both char *s and char s[128] can be used as an address, the reader must beware there is a fundamental difference between them.

(4). The assert macro: Most Unix-like systems, including Linux, support an **assert(condition)** macro, which can be used in C programs to check whether a specified condition is met or not. If the condition expression evaluates to FALSE (0), the program will abort with an error message. As an example, consider the following code segments, in which the mysum() function is designed to return the sum of an integer array (pointed by int *ptr) of n<=128 elements. Since the function is called from other code of a program, which may pass in invalid parameters, we must ensure the pointer ptr is not NULL and the array size n does not exceed the LIMIT. These can be done by including assert() statements at the entry point of a function, as shown below.

```
#define LIMIT 128
int mysum(int *ptr, int n)
{
    int i = 0, sum = 0;
    assert(ptr != NULL);     // assert ptr not NULL
    assert(n <= LIMIT);      // assert n <= LIMIT
    while(i++ < n)
       sum += *ptr++
}
```

When execution enters the mysum() function, if either **assert(condition)** fails, the function will abort with an error message.

(5). Use fprintf() and getchar() in Program Code: When writing C programs, it is often very useful to include fprintf(stderr, message) statements at key places in the program code to display expected results. Since fprintf() to stderr is unbuffered, the printed message or results will show up immediately before the next C statement is executed. If desired, the programmer may also user getchar () to stop the program flow, allowing the user to examine the execution results at that moment before continuing. We cite a simple program task to illustrate the point. A priority queue is a singly link list ordered by priority, with high priority entries in front. Entries with the same priority are ordered First-in-First-out (FIFO). Write an enqueue() function which insert an item into a priority queue by priority.

```
typedef struct entry{
        struct entry *next;
        char name[64];      // entry name
        int priority;       // entry priority
}ENTRY;

void printQ(ENTRY *queue)   // print queue contents
{
   while(queue){
      printf("[%s %d]-> ", queue->name, queue->priority);
      queue = queue->next;
   }
   printf("\n");
}

void enqueue(ENTRY **queue, ENTRY *p)
{
   ENTRY *q = *queue;
   printQ(q);          // show queue before insertion
   if (q==0 || p->priority > q->priority){ // first in queue
      *queue = p;
      p->next = q;
   }
   else{ // not first in queue; insert to the right spot
     while (q->next && p->priority <= q->priority)
            q = q->next;
     p->next = q->next;
     q->next = p;
   }
   printQ(q);          // show queue after insertion
}
```

In the above example code, if the enqueue() function code is incorrect, it may insert an entry to the wrong spot in the queue, which will cause other program code to fail if they rely on the queue contents being correct. In this case, using either assert() statements or relying on a debugger may offer very little help, since all the pointers are valid and it would be too tedious to trace a long link list in a debugger.

Instead, we print the queue before inserting a new entry and print it again after the insertion. These allow the user to see directly whether or not the queue is maintained correctly. After verifying the code works, the user may comment out the printQ() statements.

2.9 Structures in C

A structure is a composite data type containing a collection of variables or data objects. Structure types in C are defined by the **struct** keyword. Assume that we need a node structure containing the following fields.

next : a pointer to the next node structure;
key : an integer;
name : an array of 64 chars;

Such a structure can be defined as

```
struct node{
   struct node *next;
   int  key;
   char name[64];
};
```

Then, "struct node" can be used as a derived type to define variables of that type, as in

```
struct node x, *nodePtr;
```

These define x as a node structure and nodePtr a node pointer. Alternatively, we may define "struct node" as a derived type by the **typedef** statement in C.

```
typedef struct node{
   struct node *next;
   int  key;
   char name[64];
}NODE;
```

Then, NODE is a derived type, which can be used to define variables of that type, as in

```
NODE x, *nodePtr;
```

The following summarizes the properties of C structures.

(1). When defining a C structure, every field of the structure must have a type already known to the compiler, except for the self-referencing pointers. This is because pointers are always the same size, e.g. 4 bytes in 32-bit architecture. As an example, in the above NODE type, the filed next is a

```
struct node *next;
```

which is correct because the compiler knows **struct node** is a type (despite being incomplete yet) and how many bytes to allocate for the next pointer. In contrast, the following statements

```
typedef struct node{
   NODE *next;      // error
   int  key;
   char name[64];
}NODE;
```

would cause a compile-time error because the compiler does not know what is the NODE type yet, despite next is a pointer.
(2). Each C structure data object is allocated a piece of contiguous memory. The individual fields of a C structure are accessed by using the **. operator**, which identifies a specific field, as in

```
NODE x;   // x is a structure of NODE type
```

Then the individual fields of x are accessed as

x.next; which is a pointer to another NODE type object.
x.key; which is an integer
x.name; which is an array of 64 chars

At run time, each field is accessed as an **offset** from the beginning address of the structure.
(3). The size of a structure can be determined by **sizeof(struct type)**. The C compiler will calculate the size in total number of bytes of the structure. Due to memory alignment constraints, the C compiler may pad some of the fields of a structure with extra bytes. If needed, the user may define C structures with the **PACKED** attribute, which prevents the C compiler from padding the fields with extra bytes, as in

```
typedef struct node{
    struct node *next;
    int  key;
    char name[2];
 }__attribute__((packed, aligned(1))) NODE;
```

In this case, the size of the NODE structure will be 10 bytes. Without the packed attribute, it would be 12 bytes because the C compiler would pad the name field with 2 extra bytes, making every NODE object a multiple of 4 bytes for memory alignment.
(4). Assume that NODE x, y; are two structures of the same type. Rather than copying the individual fields of a structure, we can assign x to y by the C statement y = x. The compiler generated code uses the library function memncpy(&y, &x, sizeof(NODE)) to copy the entire structure.
(5). Unions in C is similar to structures. To define a union, simply replace the keyword **struct** with the keyword **union**, as in

```
union node{
    int  *ptr;      // pointer to integer
    int  ID;        // 4-byte integer
    char name[32];  // 32 chars
 }x;                // x is a union of 3 fields
```

Members in unions are accessed in exactly the same way as in structures. The major difference between structures and unions is that, whereas each member in a structure has a unique memory area, all members of a union share the same memory area, which is accessed by the attributes of the individual members. The size of a union is determined by the largest member. For example, in the union x the member name requires 32 bytes. All other members require only 4 bytes each. So the size of the union x is 32 bytes. The following C statements show how to access the individual members of a union.

```
x.ptr = 0x12345678;           // use first 4 bytes of x
x.ID = 12345;                 // use first 4 bytes of x also
strcpy(x.name, "1234567890"); // uses first 11 bytes of x
```

2.9.1 Structure and Pointers

In C, **pointers** are variables which point to other data objects, i.e. they contain the address of other data objects. In C programs, pointers are define with the *** attribute**, as in

```
TYPE *ptr;
```

which defines ptr as a pointer to a TYPE data object, where TYPE can be either a base type or a derived type, such as struct, in C. In C programming, structures are often accessed by pointers to the structures. As an example, assume that NODE is a structure type. The following C statements

```
NODE x, *p;
p = &x;
```

define x as a NODE type data object and p as a pointer to NODE objects. The statement p = &x; assigns the address of x to p, so that p points at the data object x. Then *p denotes the object x, and the members of x can be accessed as

```
(*p).name, (*p).value, (*p).next
```

Alternatively, the C language allows us to reference the members of x by using the "point at" **operator ->**, as in

```
p->name, p->value, p->next;
```

which are more convenient than the . operator. In fact, using the -> operator to access members of structures has become a standard practice in C programming.

2.9.2 Typecast in C

Typecast is a way to convert a variable from one data type to another data type by using the cast operator **(TYPE) variable.** Consider the following code segments.

```
     char *cp, c = 'a';        // c is 1 byte
     int  *ip, i = 0x12345678; // i is 4 bytes

(1). i = c;              // i = 0x00000061; lowest byte = c
(2). c = i;              // c = 0x78 (c = lowest byte of i)
(3). cp = (char *)&i;    // typecast to suppress compiler warning
(4). ip = (int *)&c;     // typecast to suppress compiler warning
(5). c = *(char *)ip;    // use ip as a char *
(6). i = *(int *)cp;     // use cp as an int *
```

Lines (1) and (2) do not need typecasting even though the assignments involve different data types. The resulting values of the assignments are shown in the comments. Lines (4) and (5) need typecasting in order to suppress compiler warnings. After the assignments of Lines (4) and (5), *cp is still a byte, which is the lowest byte of the (4-byte) integer i. *ip is an integer = 0x00000061, with the lowest byte = 'c' or 0x 61. Line (6) forces the compiler to use ip as a char *, so *(char *)ip dereferences to a single byte. Line (7) forces the compiler to use cp as an int *, so *(int *)cp dereferences to a 4-byte value, beginning from where cp points at.

Typecasting is especially useful with pointers, which allows the same pointer to point to data objects of different sizes. The following shows a more practical example of typecasting. In an **Ext2/3** file system, the contents of a directory are dir_entries defined as

```
struct dir_entry{
   int  ino;            // inode number
   int  entry_len;      // entry length in bytes
   int  name_len;       // name_len
   char name[  ]        // name_len chars
};
```

The contents of a directory consist of a linear list of dir_entries of the form

```
| ino elen nlen NAME | ino elen nlen NAME | . . .
```

in which the entries are of variable length due to the different name_len of the entries. Assume that char buf[ ] contains a list of dir_entries. The problem is how to traverse the dir_entries in buf[ ]. In order to step through the dir_entires sequentially, we define

```
struct dir_entry *dp = (struct dir_entry *)buf;  // typecasting
char *cp = buf;                      // no need for typecasting
// Use dp to access the current dir_entry;
// advance dp to next dir_entry:
cp = += dp->entry_len;       // advance cp by entry_len
dp = (struct dir_entry *)cp; // pull dp to where cp points at
```

With proper typecasting, the last two lines of C code can be simplified as

```
dp = (struct dir_entry *)((char *)dp + dp->rlen);
```

which eliminates the need for a char *cp.

2.10 Link List Processing

Structures and pointers are often used to construct and manipulate **dynamic data structures**, such as link lists, queues and trees, etc. The most basic type of dynamic data structures is the link list. In this section, we shall explain the concepts of link list, list operations, and demonstrate list processing by example programs.

2.10.1 Link Lists

Assume that nodes are structures of NODE type, as in

```
typedef struct node{
   struct node *next;   // next node pointer
   int  value;          // ID or key value
   char name[32];       // name field if needed
}NODE;
```

A (**singly) link list** is a data structure consisting of a sequence of nodes, which are linked together by the next pointers of the nodes, i.e. each node's next pointer points to a next node in the list. Link lists are represented by NODE pointers. For example, the following C statements define two link lists, denoted by list and head.

```
NODE *list, *head;         // define list and head as link lists
```

A link list is empty if it contains no nodes, i.e. an empty link list is just a NULL pointer. In C programming, the symbol **NULL** is defined by the macro (in stddef.h)

```
#define NULL (void *)0
```

as a pointer with a 0 value. For convenience, we shall use either NULL or 0 to denote the null pointer. Thus, we may initialize link lists as empty by assigning them with null pointers, as in

```
list = NULL;    // list = null pointer
head = 0;       // head = null pointer
```

2.10.2 Link List Operations

The most common types of operations on link lists are

Build: initialize and build list with a set of nodes
Traversal: step through the elements of a list
Search: search for an element by key
Insertion: insert new elements into a list
Deletion: delete an existing element from a list
Reorder: reorder a link list by (changed) element key or priority value

In the following, we shall develop C programs to demonstrate list processing.

2.10.3 Build Link List

In many C programming books, dynamic data structures are usually constructed with data objects that are dynamically allocated by the C library function **malloc().** In these cases, data objects are allocated from the program's **heap** area. This is fine, but some books seem to over emphasize the notion that dynamic data structures must use malloc(), which is false. Unlike static data structures, such as arrays, in which the number of elements is fixed, dynamic data structures consists of data objects that can be modified with ease, such as to insert new data objects, delete existing data objects and reorder the data objects, etc. It has nothing to do with where the data objects are located, which can be in either the program's data area or heap area. We shall clarify and demonstrate this point in the following example programs.

2.10.3.1 Link List in Data Area

Example Program C2.1: This program builds a link list from an array of node structures. The program consists of a type.h header file and a C2.1.c file, which contains the main() function.

(1). **type.h file**: First, we write a type.h file, which includes the standard header files, defines the constant N and the NODE type. The same type.h file will be used as an included file in all list processing example programs.

```
/********** type.h file ***********/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define N 10

typedef struct node{
  struct node *next;
  int  id;
  char name[64];

}NODE;
/******* end of type.h file *******/
```

(2). **C2.1 Program code:** The program constructs a link list and prints the link list.

```
/********************** C2.1.c file **********************/
#include "type.h"

NODE *mylist, node[N]; // in bss section of program run-time image

int printlist(NODE *p) // print list function
{
   while(p){
     printf("[%s %d]->", p->name, p->id);
     p = p->next;
```

```
   }
   printf("NULL\n");
}

int main()
{
  int i;
  NODE *p;
  for (i=0; i<N; i++){
     p = &node[i];
     sprintf(p->name, "%s%d", "node", i); // node0, node1, etc.
     p->id = i;            // used node index as ID
     p->next = p+1;        // node[i].next = &node[i+1];
  }
  node[N-1].next = 0;
  mylist = &node[0];       // mylist points to node[0]
  printlist(mylist);       // print mylist elements
}
```

(4). Explanation of the C2.1.c code

(4).1. Memory location of variables: The following diagram shows the memory location of the variables, which are all in the program's Data area. More precisely, they are in the bss section of the combined program Data area.

```
NODE *mylist,    node[N]; // in the bss section of run-time image
      -------    --0----1----2-----------------------------------9---
      |  ?  |    |next|    |    |                                |    |
      -------    |id  |    |    |                                |    |
                 |name|    |    |                                |    |
                 ---------------------------------------------------
```

(4).2. The main() function: The main() function builds a link list with node names = "node0" to "node9" and node id values=0 to 9. The link list ends with a null pointer. Note that the list pointer and list elements are all in the array of node[N], which is the program's data area.

(5). Run Results: The reader may compile and run the C2.1 program. It should print the list as

```
[node0 0]->[ndoe1 1]-> ... [node9 9]->NULL
```

2.10.3.2 Link List in Heap Area

Example C2.2: The next example program, C2.2, builds a link list in the program's **heap area**.

```
/*************** C2.2.c file *****************/
#include "type.h"

NODE *mylist, *node;    // pointers, no NODE area yet

int printlist(NODE *p){ // same as in C2.1 }
```

```
int main()
{
  int i;
  NODE *p;
  node = (NODE *)malloc(N*sizeof(NODE)); // node->N*72 bytes in HEAP
  for (i=0; i < N; i++){
     p = &node[i];          // access each NODE area in HEAP
     sprintf(p->name, "%s%d", "node",i);
     p->id = i;
     p->next = p+1;         // node[i].next = &node[i+1];
  }
  node[N-1].next = 0;
  mylist = &node[0];
  printlist(mylist);
}
```

(1). Memory location of variables: In the C2.2 program, both the variables mylist and node are NODE pointers, which are uninitialized globals. As such they are in the program's bss section, as shown in Fig. 2.21. The program does not have any memory locations for the actual node structures.

```
NODE *mylist, *node;    // pointers, no NODE area yet
```

When the program starts, it uses the statement

```
node = (NODE *)malloc(N*sizeof(NODE));
```

to allocate N*72 bytes in the program's HEAP area, which will be used as nodes of the list. Since node is a NODE pointer, which points to a memory area containing N adjacent nodes, we may regard the memory area as an array and use node[i] (i=0 to N-1) to access each node element. This is a general principle. Whenever we have a pointer pointing to a memory area containing N adjacent data objects, we may access the memory area as a sequence of array elements. As a further example, assume

```
int *p = (int *)malloc(N*sizeof(int));
```

Then the memory area pointed by p can be accessed as *(p + i) or p[i], i=0 to N-1.

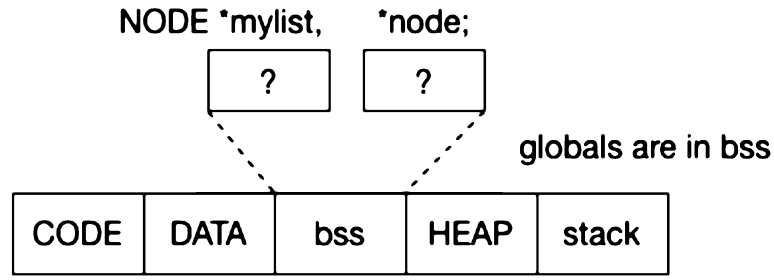


Fig. 2.21 Variables in BSS section

Example C2.3: The example program C2.3 builds a link list in the program's heap area but with individually allocated nodes.

```
/*************** C2.3.c file ****************/
#include "type.h"

NODE *mylist, *node;    // pointers, no NODE area yet

int printlist(NODE *p){ // same as in L1.c }

int main()
{
  int i;
  NODE *p, *q;
  for (i=0; i < N; i++){
     p = (NODE *)malloc(sizeof(NODE));  // allocate a node in heap
     sprintf(p->name, "%s%d", "node",i);
     p->id = i;
     p->next = 0;       // node[i].next = 0;
     if (i==0){
        mylist = q = p; // mylist -> node0; q->current node
     }
     q->next = p;
     q = p;
  }
  printlist(mylist);
}
```

2.10.4 Link List Traversal

Link list traversal is to step through the list elements in sequential order. The simplest link list traversal is to print the list elements. To do this, we may implement a generic printlist() function as follows.

```
int printlist(char *listname, NODE *list)
{
   printf("%s = ", name);
   while(list){
     printf("[%s %d]->", list->name, list->id);
     list = list->next;
  }
  printf("NULL\n");
}
```

Alternatively, the printlist() function can be implemented by recursion.

```
void rplist(NODE *p)
{
   if (p == 0){
```

```
      printf("NULL\n");
      return;
   }
   printf("[%s %d]->", p->name, p->id);
   rplist(p->next);        // recursive call to rplist(p->next)
}

int printlist(char *listname, NODE *list)
{
   printf("%s = ", name);  // print list name
   rplist(list);           // recursively print each element
}
```

In addition to printing a link list, there are many other list operations that are essentially variations of the list traversal problem. Here we cite such a problem, which is a favorite question during interviews for Computer Science jobs.

This example implements a function which computes the SUM of the node values in a link list.

```
int sum(NODE *list) // return SUM of node values in a link list
{
    int sum = 0;
    while(list){
       sum += list->value;  // add value of current node
       list = list->next;   // step to next node
    }
    return sum;
}
```

Alternatively, the sum() function can be implemented recursively, as in

```
int sum(NODE *list) // return SUM of node values in a link list
{
    if (list == 0)
       return 0;
    return list->value + sum(list->next);
}
```

Using the **? operator** of C, this can be simplified further as

```
int sum(NODE *list)
{ return (list==0)? 0: list->value + sum(list->next); }
```

When traversing a singly link list using a single node pointer, we can only access the current node but not the previous node. A simple way to remedy this problem is to use two pointers while traversing a link list. We illustrate this technique by an example.

Traversing a (singly) link list with two pointers.

```
NODE *p, *q;
p = q = list;          // both p and q start form list
while(p){
   // access current node by p AND previous node by q
   q = p;              // let q point at current node
   p = p->next;        // advance p to next node
}
```

In the above code segment, p points at the current node and q points at the previous node, if any. This allows us to access both the current and previous nodes. This technique can be used in such operations as to delete the current node, insert before the current node, etc.

2.10.5 Search Link List

The search operation searches a link list for an element with a given key, which can be either an integer value or a string. It returns a pointer to the element if it exists, or NULL if not. Assuming that the key is an integer value, we can implement a search() function as follows.

```
NODE *search(NODE *list, int key)
{
   NODE *p = list;
   while(p){
     if (p->key == key)  // found element with key
        return p;        // return node pointer
     p = p->next;        // advance p to next node
   }
   return 0;             // not found, return 0
}
```

2.10.6 Insert Operation

The insert operation inserts a new element into a link list by a specified criterion. For singly link lists, the simplest insertion is to insert to the end of a list. This amounts to traversing the list to the last element and then adding a new element to the end. We illustrate this by an example.

Example C2.4: The example program C2.4 builds a link list by inserting new nodes to the end of list.

(1) **The insertion function:** The insert() function enters a new node pointer p into a list. Since the insert operation may modify the list itself, we must pass the list parameter by reference, i.e. passing the address of the list. Normally, all elements in a link list should have unique keys. To ensure no two elements with the same key, the insert function may check this first. It should reject the insertion request if the node key already exists. We leave this as an exercise for the reader.

```
int insert(NODE **list, NODE *p) // insert p to end of *list
{
   NODE *q = *list;
   if (q == 0)          // if list is empty:
      *list = p;        //    insert p as first element
   else{                // otherwise, insert p to end of list
      while(q->next)    // step to LAST element
         q = q->next;
      q->next = p;      // let LAST element point to p
   }
   p->next = 0;         // p is the new last element.
}
```

(2) **The C2.4.c file:**

```
/***************** C2.4.c file **************/
#include "type.h"
NODE *mylist;
int main()
{
  char line[128], name[64];
  int id;
  NODE *p;
  mylist = 0;  // initialize mylist to empty list
  while(1){
     printf("enter node name and id value : ");
     fgets(line, 128, stdin);    // get an input line
     line[strlen(line)-1] = 0;   // kill \n at end
     if (line[0] == 0)           // break out on empty input line
        break;
     sscanf("%s %d", name, &id); // extract name string and id value
     p = (NODE *)malloc(sizeof(NODE));
     if (p==0) exit(-1);         // out of HEAP memory
     strcpy(p->name, name);
     p->id = id;
     insert(&mylist, p);         // insert p to list end
     printlist(mylist);
  }
}
```

2.10.7 Priority Queue

A **priority queue** is a (singly) link list ordered by priority, with high priority values in front. In a priority queue, nodes with the same priority are ordered First-In-First-Out (FIFO). We can modify the insert() function to an enqueue() function, which inserts nodes into a priority queue by priority. Assume that each node has a priority field. The following shows the enqueue() function.

```
int enqueue(NODE **queue, NODE *p) // insert p into queue by priority
{
   NODE *q = *queue;
   if (q==0 || p->priority > q->priority){
      *queue = p;
      p->next = q;
   }
   else{
      while(q->next && p->priority <= q->next->priority)
         q = q->next;
      p->next = q->next;
      q->next = p;
   }
}
```

2.10.8 Delete Operation

Given a link list and a list element key, delete the element from the list if it exists. Since the delete operation may modify the list, the list parameter must be passed by reference. Here we assume the element key is an integer. The delete() function returns the deleted node pointer, which can be used to free the node if it was dynamically allocated earlier.

```
NODE *delete(NODE **list, int key)
{
  NODE *p, *q;
  if (*list == 0)           // empty list
      return 0;             // return 0 if deletion failed
  p = *list;
  if (p->key == key){       // found element at list beginning
    *list = p->next;        // modify *list
    return p;
  }
  // element to be deleted is not the first one; try to find it
  q = p->next;
  while(q){
    if (q->key == key){
       p->next = q->next;  // delete q from list
       return q;
    }
    p = q;
    q = q->next;
  }
  return 0;                 // failed to find element
}
```

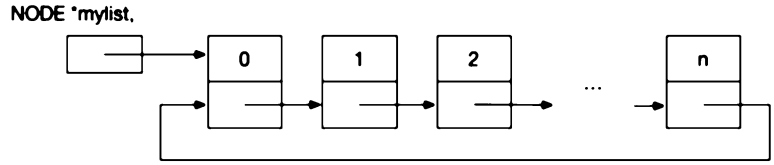


Fig. 2.22 Circular link list

2.10.9 Circular Link List

A circular link list is one in which the last element points to the first element. As usual, an empty circular link list is a NULL pointer. Figure 2.22 shows a circular link list.

When traversing a circular list, we must detect the "end of list" condition properly to terminate the traversal. Assume that list is a non-empty circular list.

```
NODE *list, *p = list;
while(p->next != list){   // stop after last element
   // access current node p;
   p = p->next;
}
```

2.10.10 Open-Ended C Structures

In the original C language, all members of a structure must be fully specified so that the size of every structure is fixed at compile time. In practice, there are situations in which the size of a structure may vary. For convenience, the C language is extended to support **open-ended structures** containing an incompletely specified field. As an example, consider the following structure, in which the last member is an array of unspecified size.

```
struct node{
    struct node *next;
    int ID;
    char name[ ];   // unspecified array size
};
```

In the above open-ended structure, name[ ] denotes an incompletely specified field, which must be the last entry. The size of an open-ended structure is determined by the specified fields. For the above example, the structure size is 8 bytes. To use such a structure, the user must allocate the needed memory for the actual structure, as in

```
struct node *sp = malloc(sizeof(struct node) + 32);
strcpy(sp->name, "this is a test string");
```

The first line allocates a memory area of 8+32 bytes, with 32 bytes intended for the field name. The next line copies a string into the name field in the allocated memory area. What if the user does not allocate memory for an open-ended structure and use them directly? The following code lines illustrate the possible consequences.

```
struct node x;                // define a variable x of size only 8 bytes
strcpy(x.name, "test");       // copy "test" into x.name field
```

In this case, x has only 8 bytes but there is no memory area for the name field. The second statement would copy the string "test" into the memory area immediately following x at run-time, overwriting whatever was in it. This may corrupt other data objects of the program, causing it to crash later.

2.10.11 Doubly Link Lists

A doubly link list, or **dlist** for short, is one in which the list elements are linked together in two directions, both forward and backward. In a dlist, each element has two pointers, denoted by next and prev, as shown below.

```
typedef struct node{
   struct node *next;     // pointer to next node
   struct node *prev;     // pointer to previous node
   // other fields, e.g. key, name, etc.
}NODE;                    // list NODE type
```

In the node structure of dlist, the next pointer points to a next node, allowing forward traversal of the dlist from left to right. The prev pointer points to a previous node, allowing backward traversal of the dlist from right to left. Unlike a singly link list, which is represented by a single NODE pointer, doubly link lists may be represented in three different ways, each with its advantages and disadvantages. In this section, we shall use example programs to show doubly link list operations, which include

(1). Build dlist by inserting new elements to the end of list.
(2). Build dlist by inserting new elements to the front of list.
(4). Print dlist in both forward and backward directions.
(5). Search dlist for an element with a given key.
(6). Delete element from dlist.

2.10.12 Doubly Link Lists Example Programs

Example Program C2.5: Doubly Link List Version 1
In the first dlist example program, we shall assume that a dlist is represented by a single NODE pointer, which points to the first list element, if any. Figure 2.23 shows such a dlist.

Since the first element has no predecessor, its prev pointer must be NULL. Likewise, the last element has no successor, its next pointer must be NULL also. An empty dlist is just a NULL pointer. The following lists the C code of the example program C2.5.

```
/********** dlist Program C2.5.c **********/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

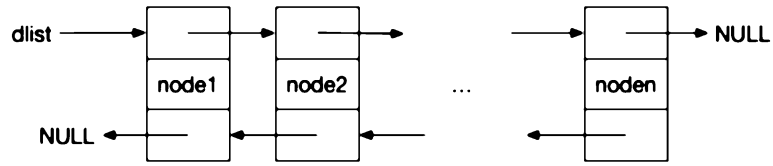


Fig. 2.23 Doubly link list version 1

```
typedef struct node{
  struct node *next;
  struct node *prev;
  int key;
}NODE;
NODE *dlist;        // dlist is a NODE pointer

int insert2end(NODE **list, int key) // insert to list END
{
  NODE *p, *q;
  //printf("insert2end: key=%d ", key);
  p = (NODE *)malloc(sizeof(NODE));
  p->key = key;
  p->next = 0;
  q = *list;
  if (q==0){           // list empty
    *list = p;
    p->next = p->prev = 0;
  }
  else{
    while(q->next)     // step to LAST element
       q = q->next;
    q->next = p;       // add p as last element
    p->prev = q;
  }
}

int insert2front(NODE **list, int key) // insert to list FRONT
{
  NODE *p, *q;
  //printf("insert2front: key=%d ", key);
  p = (NODE *)malloc(sizeof(NODE));
  p->key = key;
  p->prev = 0;         // no previous node
  q = *list;
  if (q==0){           // list empty
    *list = p;
     p->next = 0;
  }
  else{
    p->next = *list;
    q->prev = p;
    *list = p;
  }
}
```

```
void printForward(NODE *list) // print list forward
{
  printf("list forward =");
  while(list){
    printf("[%d]->", list->key);
    list = list->next;
  }
  printf("NULL\n");
}

void printBackward(NODE *list) // print list backward
{
  printf("list backward=");
  NODE *p = list;
  if (p){
     while (p->next)    // step to last element
       p = p->next;
     while(p){
       printf("[%d]->", p->key);
       p = p->prev;
     }
  }
  printf("NULL\n");
}

NODE *search(NODE *list, int key) // search for an element with a key
{
  NODE *p = list;
  while(p){
    if (p->key==key){
      printf("found %d at %x\n", key, (unsigned int)p);
      return p;
    }
    p = p->next;
  }
  return 0;
}

int delete(NODE **list, NODE *p) // delete an element pointed by p
{
  NODE *q = *list;
  if (p->next==0 && p->prev==0){        // p is the only node
    *list = 0;
  }
  else if (p->next==0 && p->prev != 0){ // last but NOT first
    p->prev->next = 0;
  }
  else if (p->prev==0 && p->next != 0){ // first but NOT last
    *list = p->next;
    p->next->prev = 0;
```

```
  }
  else{                                  // p is an interior node
    p->prev->next = p->next;
    p->next->prev = p->prev;
  }
  free(p);
}

int main()
{
  int i, key;
  NODE *p;
  printf("dlist program #1\n");

  printf("insert to END\n");
  dlist = 0;                    // initialize dlist to empty
  for (i=0; i<8; i++){
    insert2end(&dlist, i);
  }
  printForward(dlist);
  printBackward(dlist);

  printf("insert to FRONT\n");
  dlist = 0;                    // initialize dlist to empty
  for (i=0; i<8; i++){
    insert2front(&dlist, i);
  }
  printForward(dlist);
  printBackward(dlist);

  printf("test delete\n");
   while(1){
    printf("enter a key to delete: ");
    scanf("%d", &key);
    if (key < 0) exit(0);      // exit if negative key
    p = search(dlist, key);
    if (p==0){
       printf("key %d not found\n", key);
       continue;
    }
    delete(&dlist, p);
    printForward(dlist);
    printBackward(dlist);
  }
}
```

Figure 2.24 show the outputs of running the Example program C2.5. First, it constructs a dlist by inserting new elements to the list end, and prints the resulting list in both forward and backward directions. Next, it constructs a dlist by inserting new elements to the list front, and prints the resulting

```
dlist program #1
insert to END
list forward =[0]->[1]->[2]->[3]->[4]->[5]->[6]->[7]->NULL
list backward=[7]->[6]->[5]->[4]->[3]->[2]->[1]->[0]->NULL
insert to FRONT
list forward =[7]->[6]->[5]->[4]->[3]->[2]->[1]->[0]->NULL
list backward=[0]->[1]->[2]->[3]->[4]->[5]->[6]->[7]->NULL
test delete
enter a key to delete: 0
found 0 at 9567088
list forward =[7]->[6]->[5]->[4]->[3]->[2]->[1]->NULL
list backward=[1]->[2]->[3]->[4]->[5]->[6]->[7]->NULL
enter a key to delete: 7
found 7 at 95670f8
list forward =[6]->[5]->[4]->[3]->[2]->[1]->NULL
list backward=[1]->[2]->[3]->[4]->[5]->[6]->NULL
enter a key to delete: 4
found 4 at 95670c8
list forward =[6]->[5]->[3]->[2]->[1]->NULL
list backward=[1]->[2]->[3]->[5]->[6]->NULL
enter a key to delete:
```

Fig. 2.24 Outputs of example program C2.5

list in both directions. Then, it shows how to search for an element with a given key. Lastly, it shows how to delete elements from the list, and print the list after each delete operation to verify the results.

Discussions of Example C2.5: The primary advantage of the first version of dlist is that it only requires a single NODE pointer to represent a dlist. The disadvantages of this kind of dlist are:

(1). It is hard to access the last element of the list. In order to access the last element, as in the cases of inserting to the list end and print the list backward, we must use the next pointer to step to the last element first, which is time-consuming if the list has many elements.
(2). When inserting a new element, we must detect and handle the various cases of whether the list is empty or the new element is the first or last element, etc. These make the code non-uniform, which does not fully realize the capability of doubly link lists.
(3). Similarly, when deleting an element, we must detect and handle the different cases also.

As can be seen, the complexity of the insertion/deletion code stems mainly from two cases:

. the list is empty, in which case we must insert the first element.
. the inserted/deleted node is the first or last node, in which case we must modify the list pointer.

Alternatively, a doubly link list may be regarded as two singly link lists merged into one. Accordingly, we may represent doubly link lists by a listhead structure containing two pointers, as in

```
typedef struct listhead{
      struct node *next;       // head node pointer
      struct node *prev;       // tail node pointer
  }DLIST;                      // doubly link list type
  DLIST dlist;                 // dlsit is a structure
  dlist.next = dlist.prev = 0; // initialize dlist as empty
```

Figure 2.25 show the second version of doubly link lists.

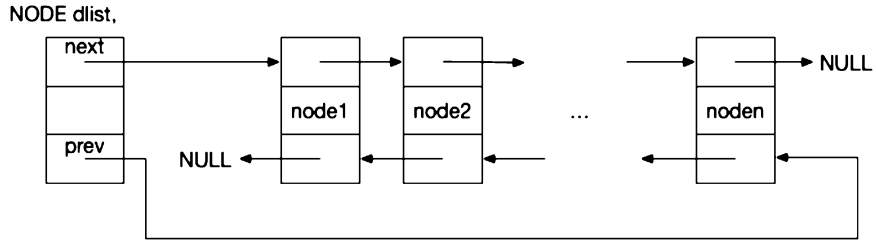


Fig. 2.25 Doubly link list version 2

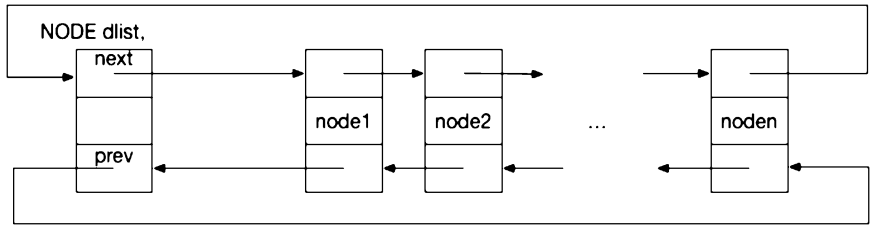


Fig. 2.26 Doubly link list version 3

The only advantage of using a list head structure is that it allows direct access to the last element via the list's prev pointer. The disadvantages are exactly the same as those in the first version of dlist. Specifically,

(1) Since dlist is a structure, it must be passed by reference to all list operation functions.
(2) In both the insertion and deletion functions, we must detect and handle the various cases of whether the list is empty, the element is the first or last element, etc. In these cases, doubly link lists are no better than singly link list in terms of processing time.

In order to realize the capability of doubly link lists, we need a better way to represent doubly link lists. In the next example, we shall define doubly lists as follows. A dlist is represented by a listhead, which is a NODE structure but a dlist is initialized with both pointers to the listhead, as in

```
NODE dlist;                          //  dlist is a NODE structure
dlist.next = dlist.prev = &dlist;    //  initialize both pointers to the NODE
                                         structure
```

Such a dlist may be regarded as two circular link lists merged into one, with the listhead as the initial dummy element. Figure 2.26 shows such a doubly link list.

Since there are no null pointers in the Version 3 dlist, every node can be treated as an interior node, which greatly simplifies the list operation code. We demonstrate this kind of dlist by an example.

Example Program C2.6: This example program assumes a dlist is represented by a NODE structure, which is initialized with both pointers to the NODE structure itself.

```
/************** C2.6.c Program Code *************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
typedef struct node{
  struct node *next;
  struct node *prev;
  int key;
}NODE;

NODE dlist; // dlist is NODE struct using only next & prev pointers

int insert2end(NODE *list, int key)
{
  NODE *p, *q;
  //printf("insert2end: key=%d\n", key);
  p = (NODE *)malloc(sizeof(NODE));
  p->key = key;
  p->prev = 0;

  q = list->prev;      // to LAST element
  p->next = q->next;
  q->next->prev = p;
  q->next = p;
  p->prev = q;
}

int insert2front(NODE *list, int key)
{
  NODE *p, *q;
  //printf("insertFront key=%d\n", key);
  p = (NODE *)malloc(sizeof(NODE));
  p->key = key;
  p->prev = 0;

  q = list->next;     // to first element
  p->prev = q->prev;
  q->prev->next = p;
  q->prev = p;
  p->next = q;
}

void printForward(NODE *list)
{
  NODE *p = list->next;    // use dlist's next pointer
  printf("list forward =");
  while(p != list){        // detect end of list
    printf("[%d]->", p->key);
    p = p->next;
  }
  printf("NULL\n");
}
```

```
void printBackward(NODE *list)
{
  printf("list backward=");
  NODE *p = list->prev;    // use dlist's prev pointer
  while(p != list){        // detect end of list
    printf("[%d]->", p->key);
    p = p->prev;
  }
  printf("NULL\n");
}

NODE *search(NODE *list, int key)
{
  NODE *p = list->next;
  while(p != list){        // detect end of list
    if (p->key==key){
      printf("found %d at %x\n", key, (unsigned int)p);
      return p;
    }
    p = p->next;
  }
  return 0;
}

int delete(NODE *list, NODE *p)
{
  p->prev->next = p->next;
  p->next->prev = p->prev;
  free(p);
}

int main()
{
  int i, key;
  NODE *p;
  printf("dlist program #3\n");

  printf("insert to END\n");
  dlist.next = dlist.prev = &dlist; // empty dlist
  for (i=0; i<8; i++){
    insert2end(&dlist, i);
  }
  printForward(&dlist);
  printBackward(&dlist);

  printf("insert to front\n");
  dlist.next = dlist.prev = &dlist; // empty dlist to begin
  for (i=0; i<8; i++){
    insert2front(&dlist, i);
  }
```

```
  printForward(&dlist);
  printBackward(&dlist);

  printf("do deletion\n");
  while(1){
    printf("enter key to delete: ");
    scanf("%d", &key);
    if (key < 0) exit(0);    // exit if key negative
    p = search(&dlist, key);
    if (p==0){
       printf("key %d not found\n", key);
       continue;
    }
    delete(&dlist, p);
    printForward(&dlist);
    printBackward(&dlist);
  }
}
```

Figure 2.27 shows the outputs of running the example program C2.6. Although the outputs are identical to those of Fig. 2.24, their processing codes are very different.

The primary advantages of the Example program C2.6 are as follows. Since every node can be treated as an interior node, it is no longer necessary to detect and handle the various cases of empty list, first and last elements, etc. As a result, both insertion and deletion operations are greatly simplified. The only modification needed is to detect the end of list condition during search or list traversal by the code segment

```
NODE *p = list.next;
while (p != &list){
  p = p->next;
}
```

```
dlist program #3
insert to END
list forward =[0]->[1]->[2]->[3]->[4]->[5]->[6]->[7]->NULL
list backward=[7]->[6]->[5]->[4]->[3]->[2]->[1]->[0]->NULL
insert to front
list forward =[7]->[6]->[5]->[4]->[3]->[2]->[1]->[0]->NULL
list backward=[0]->[1]->[2]->[3]->[4]->[5]->[6]->[7]->NULL
do deletion
enter key to delete: 0
found 0 at 8a71088
list forward =[7]->[6]->[5]->[4]->[3]->[2]->[1]->NULL
list backward=[1]->[2]->[3]->[4]->[5]->[6]->[7]->NULL
enter key to delete: 7
found 7 at 8a710f8
list forward =[6]->[5]->[4]->[3]->[2]->[1]->NULL
list backward=[1]->[2]->[3]->[4]->[5]->[6]->NULL
enter key to delete: 4
found 4 at 8a710c8
list forward =[6]->[5]->[3]->[2]->[1]->NULL
list backward=[1]->[2]->[3]->[5]->[6]->NULL
enter key to delete:
```

Fig. 2.27 Outputs of example program C2.6

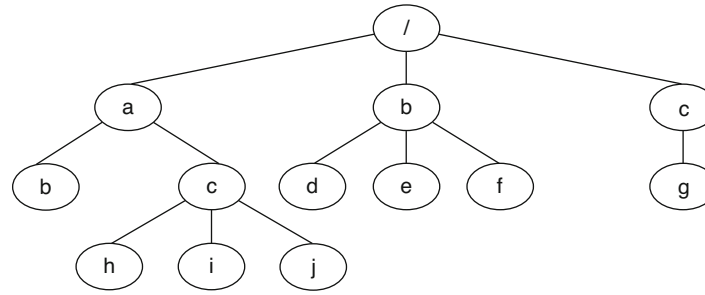


Fig. 2.28 A general tree

2.11 Trees

A **tree** is a dynamic data structure composed of multi-levels of linked lists. As a data structure, a tree is defined as a node, which itself consists of a value together with a list of references to other nodes. Symbolically, a tree is defined recursively as

```
node: value [&node[1], ..., &node[k]]
```

where each node[i] is itself a (possibly empty) tree. The very first node of a tree is called the **root** of the tree. Each node is called a **parent** node if it points to a list of other nodes, which are called the **children** nodes of the parent node. In a tree, each node has a unique parent, but each node may have a variable number of children nodes, including none. A tree can be represented by a diagram, which is usually drawn upside-down, with the root node at the top. Figure 2.28 shows a general tree.

2.12 Binary Tree

The simplest kind of tree is the **binary tree**, in which each node has two pointers, denoted by left and right. We may define the node type containing a single key value for binary trees as

```
typedef struct node{
        int   key;
        struct node *left;
        struct node *right;
}NODE;
```

Every general tree can be implemented as a binary tree. We shall demonstrate this in Sect. 2.13.

2.12.1 Binary Search Tree

A **Binary Search Tree (BST)** is a binary tree with the following properties:

. All node keys are distinct, i.e. the tree contains no nodes with duplicated keys.
. The left subtree of a node contains only nodes with keys less than the node's key.
. The right subtree of a node contains only nodes with keys greater than the node's key.
. Each of the left and right subtrees is also a binary search tree.

As an example, Fig. 2.29 shows a binary search tree.

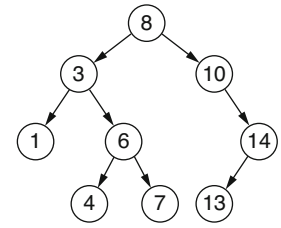


Fig. 2.29 A binary search tree

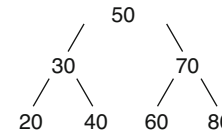
Binary Search Tree provides an ordering among the node keys so that operations such as finding the minimum, maximum keys and search for a given key can be done quickly. The search depth of a BST depends on the shape of the tree. If a binary tree is balanced, i.e. every node has two children nodes, the search depth is log2(n) for a tree with n nodes. For unbalanced BST, the worst search depth would be n, if the nodes are all to the left or all to the right.

2.12.2 Build a Binary Search Tree (BST)

Example Program C2.7: Build a Binary Search Tree

```
/************ C2.7.c file ************/
#include <stdio.h>
#include <stdlib.h>
typedef struct node{
    int key;
    struct node *left, *right;
}NODE;
#define N 7
int nodeValue[N] = {50,30,20,40,70,60,80};
// create a new node
NODE *new_node(int key)
{
    NODE *node = (NODE *)malloc(sizeof(NODE));
    node->key = key;
    node->left = node->right = NULL;
    return node;
}
// insert a new node with given key into BST
NODE *insert(NODE *node, int key)
{
    if (node == NULL)
       return new_node(key);
    if (key < node->key)
       node->left  = insert(node->left, key);
    else if (key > node->key)
       node->right = insert(node->right, key);
    return node;
}
```

Fig. 2.30 A binary search tree




```
int main()
{
  int i;
  NODE *root = NULL;
  root = insert(root, nodeValue[0]);
  for (i=1; i<N; i++){
      insert(root, nodeVlaue[i]);
  }
}
```

Figure 2.30 shows the BST generated by the Example Program C2.7.

2.12.3 Binary Tree Traversal Algorithms

From elementary data structure courses, the reader should have learned the following binary tree traversal algorithms.

. **Pre-order Traversal:** node; node.left; node.ight
. **In-order traversal:** node.left; node; node.right
. **Post-order traversal:** node.left; node.right; node

2.12.4 Depth-First Traversal Algorithms

All the above algorithms belong to **Depth-First (DF)** search/traversal algorithms, which use a stack for back-tracking. As such, they can be implemented naturally by recursion. As an example, search a BST for a given key can be implemented by rercusion, which is basically an in-order traversal of the BST.

```
/******** Search BST for a give key ********/
NODE *search(NODE *t, int key)
{
    if (t == NULL || t->key == key)
       return t;
    if (key < t->key)                // key is less than node key
       return search(t->left, key);
    else
       return search(t->right, key); // key is greater than node key
}
```

2.12.5 Breadth-First Traversal Algorithms

A tree can also be traversed by **Breadth-First (BF) algorithms**, which use a **queue** to store the yet to be traversal parts of the tree. When applying BF traversal algorithm to a tree, we may print the tree level by level, as shown by the next example program.

Example Program C2.8: Print Binary Tree by Levels

```
/***** C2.8.c file: print binary tree by levels *****/
#include <stdio.h>
#include <stdlib.h>

typedef struct node{
    struct node *left;
    int key;
    struct node *right;
}NODE;

typedef struct qe{       // queue element structure
    struct qe    *next;  // queue pointer
    struct node  *node;  // queue contents
}QE;                     // queue element type

int enqueue(QE **queue, NODE *node)
{
  QE *q = *queue;
  QE *r = (QE *)malloc(sizeof(QE));
  r->node = node;
  if (q == 0)
     *queue = r;
  else{
    while (q->next)
       q = q->next;
    q->next = r;
  }
  r->next = 0;
}

NODE *dequeue(QE **queue)
{
  QE *q = *queue;
  if (q)
     *queue = q->next;
  return q->node;
}

int qlength(QE *queue)
{
  int n = 0;
```

```
  while (queue){
    n++;
    queue = queue->next;
  }
  return n;
}

// print a binary tree by levels, each level on a line
void printLevel(NODE *root)
{
  int nodeCount;
  if (root == NULL) return;
  QE queue = 0;              // create a FIFO queue
  enqueue(&queue, root);     // start with root
  while(1){
     nodeCount = qlength(queue);
     if (nodeCount == 0) break;
     // dequeue nodes of current level, enqueue nodes of next level
     while (nodeCount > 0){
        NODE *node = dequeue(&queue);
        printf("%d ", node->key);
        if (node->left != NULL)
           enqueue(&queue, node->left);
        if (node->right != NULL)
           enqueue(&queue, node->right);
        nodeCount--;
     }
     printf("\n");
  }
}
NODE *newNode(int key) // create a new node
{
  NODE *t = (NODE *)malloc(sizeof(NODE));
  t->key = key;
  t->left = NULL;
  t->right = NULL;
  return t;
}
int main()  // driver program to test printLevel()
{
  queue = 0;
  // create a simple binary tree
  NODE *root = newNode(1);
  root->left = newNode(2);
  root->right = newNode(3);
  root->left->left = newNode(4);
  root->left->right = newNode(5);
  root->right->leftt = newNode(6);
  root->right->right = 0; // right child = 0
  printLevel(root);
  return 0;
}
```

In the example program C2.8, each queue element has a next pointer and a NODE pointer to the node in the tree. The tree will be traversed but not altered in anyway; only its nodes (pointers) will be used in queue elements, which are entered into and removed from the queue. The printLevel() function starts with the root node in the queue. For each iteration, it dequeues a node of the current level, prints its key and enters the left child (if any), followed by the right child (if any), into the queue. The iteration ends when the queue becomes empty. For the simple binary tree constructed in main(), the program outputs are

```
1
2 3
4 5 6
```

which print each level of the tree on a separate line.

2.13 Programming Project: Unix/Linux File System Tree Simulator

Summarizing the background information on C structures, link list processing and binary trees, we are ready to integrate these concepts and techniques into a programming project to solve a practical problem. The programming project is to design and implement a Unix/Linux file system tree simulator.

2.13.1 Unix/Linux File System Tree

The logical organization of a Unix file system is a general tree, as shown in the Fig. 2.28. Linux file systems are organized in the same way, so the discussions here apply to Linux files systems also. The file system tree is usually drawn upside down, with the root node / at the top. For simplicity, we shall assume that the file system contains only directories (DIRs) and regular FILEs, i.e. no special files, which are I/O devices. A DIR node may have a variable number of **children nodes**. Children nodes of the same parent are called **siblings**. In a Unix/Linux file system, each node is represented by a unique **pathname** of the form /a/b/c or a/b/c. A pathname is **absolute** if it begins with /, indicating that it starts from the root. Otherwise, it is relative to the **Current Working Directory** (CWD).

2.13.2 Implement General Tree by Binary Tree

A general tree can be implemented as a binary tree. For each node, let childPtr point to the oldest child, and siblingPtr point to the oldest sibling. For convenience, each node also has a parentPtr pointing to its parent node. For the **root node**, both parentPtr and siblingPtr point to itself. As an example, Fig. 2.31 shows a binary tree which is equivalent to the general tree of Fig. 2.28. In Fig. 2.31, thin lines represent childPtr pointers and thick lines represent siblingPtr. For clarify, NULL pointers are not shown.

2.13.3 Project Specification and Requirements

The project is to design and implement a C program to simulate the Unix/Linux file system tree. The program should work as follows.

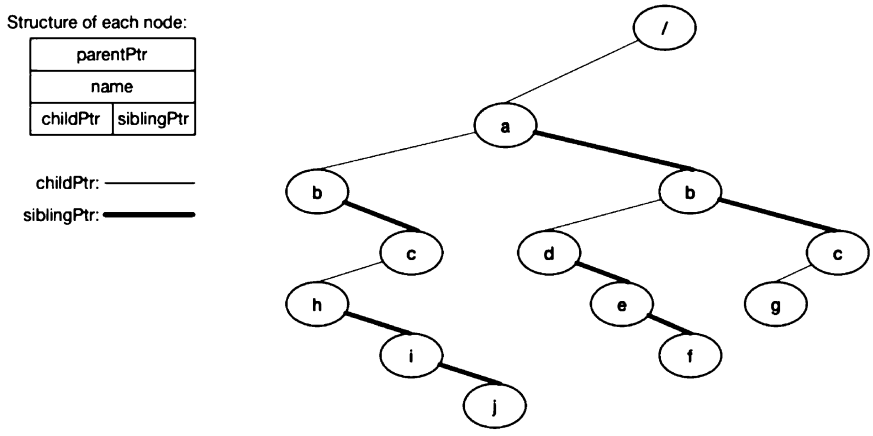


Fig. 2.31 Implementation of general tree by binary tree

(1). Start with a / node, which is also the Current Working Directory (CWD).
(2). Prompt the user for a command. Valid commands are:
mkdir, rmdir, cd, ls, pwd, creat, rm, save, reload, menu, quit
(3). Execute the command, with appropriate tracing messages.
(4). Repeat (2) until the "quit" command, which terminates the program.

2.13.4 Commands Specification

mkdir pathname :make a new directory for a given pathname
rmdir pathname :remove the directory, if it is empty.
cd [pathname] :change CWD to pathname, or to / if no pathname.
ls [pathname] :list the directory contents of pathname or CWD
pwd :print the (absolute) pathname of CWD
creat pathname :create a FILE node.
rm pathname :remove the FILE node.
save filename :save the current file system tree as a file
reload filename :construct a file system tree from a file
menu :show a menu of valid commands
quit :save the file system tree, then terminate the program.

2.13.5 Program Organization

There are many ways to design and implement such a simulator program. The following outlines the suggested organization of the program.

(1) **NODE type:** Define a C structure for the NODE type containing

64 chars : name string of the node;
char : node type: 'D' for directory or 'F' for file
node pointers : *childPtr, *siblingPtr, *parentPtr;

(2) **Global Variables:**

```
NODE *root, *cwd;                  // root and CWD pointers
char line[128];                    // user input command line
char command[16], pathname[64];    // command and pathname strings
char dname[64], bname[64];         // dirname and basename string holders
(Others as needed)
```

(3) **The main() function:** The main function of the program can be sketched as follows.

```
int main()
{
  initialize();   //initialize root node of the file system tree
  while(1){
    get user input line = [command pathname];
    identify the command;
    execute the command;
    break if command="quit";
  }
```

(4) **Get user inputs:** Assume that each user input line contains a command with an optional pathname. The reader may use scanf() to read user inputs from stdin. A better technique is as follows

```
fgets(line, 128, stdin);  // get at most 128 chars from stdin
line[strlen(line)-1] = 0; // kill \n at end of line
sscanf(line, "%s %s", command, pathname);
```

The sscanf() function extracts items from the line[ ] area by format, which can be either chars, strings or integers. It is therefore more flexible than scanf().

(5) **Identify command:** Since each command is a string, most readers would probably try to identify the command by using strcmp() in a series of if-else-if statements, as in

```
if (!strcmp(command, "mkdir)
    mkdir(pathname);
else if (!strcmp(command, "rmdir"
    rmdir(pathname);
else if . . .
```

This requires many lines of string comparisons. A better technique is to use a command table containing command strings (pointers), which ends with a NULL pointer.

```
char *cmd[] = {"mkdir", "rmdir", "ls", "cd", "pwd", "creat", "rm",
               "reload", "save", "menu", "quit", NULL};
```

For a given command, search the command table for the command string and return its index, as shown by the following findCmd() function.

```
int findCmd(char *command)
{
   int i = 0;
   while(cmd[i]){
     if (!strcmp(command, cmd[i]))
         return i;  // found command: return index i
     i++;
   }
   return -1;       // not found: return -1
}
```

As an example, for the command = "creat",

```
int index = findCmd("creat");
```

returns the index 5, which can be used to invoke a corresponding creat() function.

(6) **The main() function:** Assume that, for each command, we have written a corresponding action function, e.g. mkdir(), rmdir(), ls(), cd(), etc. The main() function can be refined as shown below.

```
int main()
{
  int index;
  char line[128], command[16], pathname[64];
  initialize();   //initialize root node of the file system tree
  while(1){
    printf("input a commad line : ");
    fgets(line,128,stdin);
    line[strlen(line)-1] = 0;
    sscanf(line, "%s %s", command, pathname);
    index = fidnCmd(command);
    switch(index){
        case 0 : mkdir(pathname);    break;
        case 1 : rmdir(pathname);    break;
        case 2 : ls(pathname);       break;
        etc.
       default: printf("invalid command %s\n", command);
    }
  }
}
```

The program uses the command index as different cases in a switch table. This works fine if the number of commands is small. For large number of commands, it would be preferable to use a table of function pointers. Assume that we have implemented the command functions

```
int mkdir(char *pathname){.........}
int rmdir(char *pathname){.........}
etc.
```

Define a **table of function pointers** containing function names in the same order as their indices, as in

```
                                    0     1     2  3  4    5    6
int (*fptr[ ])(char *)={(int (*)())mkdir,rmdir,ls,cd,pwd,creat,rm,. .};
```

The linker will populate the table with the entry addresses of the functions. Given a command index, we may call the corresponding function directly, passing as parameter pathname to the function, as in

```
int r  =  fptr[index](pathname);
```

2.13.6 Command Algorithms

Each user command invokes a corresponding action function, which implements the command. The following describes the algorithms of the action functions

mkdir pathname

(1). Break up pathname into dirname and basename, e.g.
ABSOLUTE: pathname=/a/b/c/d. Then dirname=/a/b/c, basename=d
RELATIVE: pathname= a/b/c/d. Then dirname=a/b/c, basename=d
(2). Search for the dirname node:
ASSOLUTE pathname: start from /
RELATIVE pathname: start from CWD.
if nonexist : error messages and return FAIL
if exist but not DIR: error message and return FAIL

(3). (dirname exists and is a DIR):
Search for basename in (under) the dirname node:
if already exists: error message and return FAIL;
ADD a new DIR node under dirname;
Return SUCCESS

rmdir pathname

(1). if pathname is absolute, start = /

else start = CWD, which points CWD node

(2). search for pathname node:
tokenize pathname into components strings;
begin from start, search for each component;
return ERROR if fails
(3). pathname exists:
check it's a DIR type;
check DIR is empty; can't rmdir if NOT empty;
(4). delete node from parent's child list;

creat pathname
SAME AS mkdir except the node type is 'F'

rm pathname
SAME AS rmdir except check it's a file, no need to check for EMPTY.

cd pathname
(1). find pathname node;
(2). check it's a DIR;
(3). change CWD to point at DIR

ls pathname
(1). find pathname node
(2). list all children nodes in the form of [TYPE NAME] [TYPE NAME] . . .

pwd
Start from CWD, implement pwd by recursion:

(1). Save the name (string) of the current node
(2). Follow parentPtr to the parent node until the root node;
(3). Print the names by adding / to each name string

Save Tree to a FILE The simulator program builds a file system tree in memory. When the program exits, all memory contents will be lost. Rather than building a new tree every time, we may save the current tree as a file, which can be used to restore the tree later. In order to save the current tree as a file, we need to open a file for write mode. The following code segment shows how to open a file stream for writing, and then write (text) lines to it.

```
FILE *fp = fopen("myfile", "w+");   // fopen a FILE stream for WRITE
fprintf(fp, "%c %s", 'D', "string\n"); // print a line to file
fclose(fp);                         // close FILE stream when done
```

save(filename) This function save the absolute pathnames of the tree as (text) lines in a file opened for WRITE. Assume that the file system tree is as shown in Fig. 2.32.

```
type   pathname
----   -------
 D       /
 D       /A
 F       /A/x
 F       /A/y
 D       /B
 F       /B/z
 D       /C
 D       /C/E
```

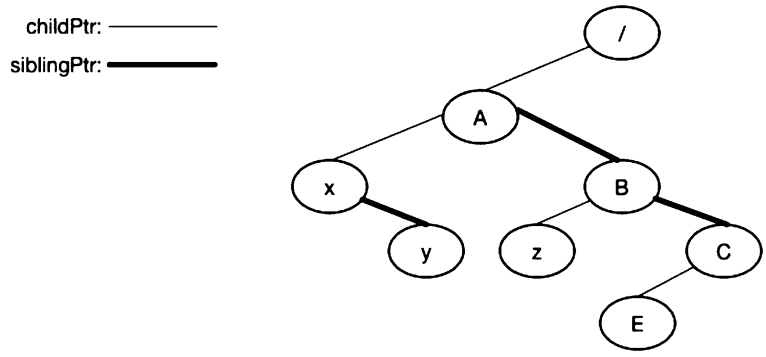


Fig. 2.32 A file system tree where uppercase names A, B, C, E are DIRs and lowercase names x, y, z are FILEs. The tree can be represented by the (text) lines

The pathnames are generated by **PRE-ORDER traversal** of a binary tree:

```
print node        name;    // current node
print node.left   name;    // left pointer = childPtr
print node.right name;     // right pointer = siblingPtr
```

Each print function prints the absolute pathname of a node, which is essentially the same as pwd(). Since the root node always exists, it can be omitted from the save file.

reload(filename) The function reconstructs a tree from a file. First, initialize the tree as empty, i.e. with only the root node. Then read each line of the file. If the line contains "D pathname", call

mkdir(pathname) to make a directory.

If the line contains "F pathname", call

creat(pathname) to create a file.

These will reconstruct the tree saved earlier.

Quit Command save the current tree to a file. Then terminate the program execution.

On subsequent runs of the simulator program, the user may use the reload command to restore the tree saved earlier.

(8). Additional Programming HELP

(8).1. Tokenize pathname into components: Given a pathname, e.g. "/a/b/c/d", the following code segment shows how to tokenize pathname into component strings.

```
int tokenize(char *pathname)
{
  char *s;
  s = strtok(path, "/");  // first call to strtok()
  while(s){
     printf("%s ", s);
     s = strtok(0, "/");  // call strtok() until it returns NULL
  }
}
```

The strtok() function divides a string into substrings by the specified delimiter char "/". The substrings reside in the original string, thus destroying the original string. In order to preserve the original pathname, the user must pass a copy of pathname to the strtok() function. In order to access the tokenized substrings, the user must ensure that they are accessible by one of the following schemes.

. The copied pathname is a global variable, e.g. char path[128], which contains the tokenized substrings.
. If the copied pathname path[ ] is local in a function, access the substrings only in the function

(8).2. dir_name and base_name: For the simulator program, it is also often necessary to decompose a pathname into dir_name, and base_name, where dir_name is the directory part of pathname and base_name is the last component of pathname. As an example, if pathname="/a/b/c", then dir_name="/a/b" and base_name="c". This can be done by the library functions dirname() and basename(), both of which destroy the pathname also. The following code segments show how to decompose a pathname into dir_name and base_name.

```
#include <libgen.h>
char dname[64], bname[64]; // for decomposed dir_name and base_name

int dbname(char *pathname)
{
  char temp[128];  // dirname(), basename() destroy original pathname
  strcpy(temp, pathname);
  strcpy(dname, dirname(temp));
  strcpy(temp, pathname);
  strcpy(bname, basename(temp));
}
```

2.13.7 Sample Solution

Sample solution of the programming project is available online at the book's website for download. Source code of the project solution is available to instructors upon request from the author.

2.14 Summary

This chapter covers the background information needed for systems programming. It introduces several GUI based text editors, such as vim, gedit and EMACS, to allow readers to edit files. It shows how to use the EMACS editor in both command and GUI mode to edit, compile and execute C programs. It explains program development steps. These include the compile-link steps of GCC, static and dynamic linking, format and contents of binary executable files, program execution and termination. It explains function call conventions and run-time stack usage in detail. These include parameter passing, local variables and stack frames. It also shows how to link C programs with assembly code. It covers the GNU make facility and shows how to write makefiles by examples. It shows how to use the GDB debugger to debug C programs. It points out the common errors in C programs and suggests ways to prevent such errors during program development. Then it covers advanced programming techniques. It describes structures and pointer in C. It covers link lists and list processing by detailed

examples. It covers binary trees and tree traversal algorithms. The chapter cumulates with a programming project, which is for the reader to implement a binary tree to simulate operations in the Unix/Linux file system tree. The project starts with a single root directory node. It supports mkdir, rmdir, creat, rm, cd, pwd, ls operations, saving the file system tree as a file and restoring the file system tree from saved file. The project allows the reader to apply and practice the programming techniques of tokenizing strings, parsing user commands and using function pointers to invoke functions for command processing.

Problems

1. Refer to the assembly code generated by GCC in Sect. 2.5.1. While in the function A(int x, int y), show how to access parameters x and y:
2. Refer to Example 1 in Sect. 2.5.2, Write assembly code functions to get CPU's ebx, ecx, edx, esi and edi registers.
3. Assume: The Intel x86 CPU registers CX=N, BX points to a memory area containing N integers. The following code segments implements a simple loop in assembly, which loads AX with each successive element of the integer array.

```
loop:   movl  (%ebx), %eax   # AX = *BX (consider BX as int * in C)
        addl  $4, %ebx       # ++BX
        subl  $1, %ecx       # CX--;
        jne                  # jump to loop if CX NON-zero
```

   Implement a function int Asum(int *a, int N) in assembly, which computes and returns the sum of N integers in the array int a[N]. Test your Asum() function by

```
int a[100], N = 100;
int main()
{
    int i, sum;
    for (i=0; i<N; i++) // set a[] values to 1 to 100
        a[i] = i+1;
    sum = Asum(a, N);   // a[] is an int array, a is int *
    printf("sum = %d\n", sum);
}
```

   The value of sum should be 5050.
4. Every C program must have a main() function.
   (1) Why?
   (2) The following program consists of a t.c file in C and a ts.s file in 32-bit assembly. The program's main() function is written in assssembly, which calls mymain() in C. The program is compile-linked by

```
gcc -m32 ts.s t.c
```

   It is run as **a.out one two three.**

```
# ******* ts.s file *******
       .global main, mymain    # int mymain() is in C
main:   pushl %ebp
        movl  %esp, %ebp
(2).1:  WRITE assembly CODE TO call mymain(argc, argv, env)
        call mymain
        leave
        ret

/*********** t.c file of a C program *************/
int mymain(int argc, char *argv[], char *env[ ])
{
    int i;
    printf("argc=%d\n", argc);
    i = 0;
    while(argv[i]){
      printf("argv[%d] = %s\n", i, argv[i]);
      i++;
    }
(2).2: WRITE C code to print all env[ ] strings
 }
```

Complete the missing code at the labels (2).1 and (2).2 to make the program work.

5. In the Makefile Example 5 of Sect. 2.7.3, the suffix rule

```
.s.o: # build each .o file if its .s file has changed
       ${AS} -a $< -o $*.o > $*.map
```

$(AS) –a tells the assembler to generate an assembly listing, which is redirected to a map file. Assume that the PMTX Kernel directory contains the following assembly code files:

```
entry.s, init.s, traps.s, ts.s
```

What are the file names of the resulting .o and .map files?

6. Refer to the Insertion function in Sect. 2.10.6. Rewrite the insert() function to ensure there are no duplicated keys in the link list.
7. Refer to the delete() function in Sect. 2.10.8. Rewrite the delete() function as

```
NODE *delete(NODE **list, NODE *p)
```

which deletes a given node pointed by p from list..

8. The following diagram shows a Multilevel Priority Queue (MPQ).
   (1) Design a data structure for a MPQ.
   (2) Design and implement an algorithm to insert a queue element with a priority between 1 and n into a MPQ.
9. Assume that a Version 1 doubly link list is represented by a single NODE pointer, as in the Example Program C2.5 in Sect. 2.10.12.
   (1). Implement a insertAfter(NODE **dlist, NODE *p, int key) function, which inserts a new node with a given key into a dlist AFTER a given node p. If p does not exist, e.g. p=0, insert to the list end.

(2). Implement a insertBefore(NODE **dlist, NODE *p, int key) function, which inserts a new node with a given key BEFORE the node p. If p does not exist, e.g. p=0, insert to the list front.
(3). Write a complete C program, which builds a dlist by the insertAfter() and insertBefore() functions.

10. Assume that a Version 2 doubly link list is represented by a list head structure.
(1). Implement a insertAfter(NODE *dlist, NODE *p, int key) function, which inserts a new node with a given key into a dlist AFTER a given node p. If p does not exist, insert to the list end.
(2). Implement a insertBefore(NODE *dlist, NODE *p, int key) function, which inserts a new node with a given key BEFORE the node p. If p does not exist, insert to list front.
(3). Write a complete C program, which builds a dlist by the insertAfter() and insertBefore() functions.
11. Assume that a doubly link list is represented by an initialized listhead as in Example Program C2.6.
(1). Implement a insertBefore(NODE *dlist, NODE *p, NODE *new) function, which inserts a new node into a dlist BEFORE an existing node p.
(2). Implement a insertAfter(NODE *dlist, NODE *p, NODE *new) function, which inserts a new node into a dlist AFTER an existing node p.
(3). Write a complete C program to build a dlist by using the insertAfter() and insertBefore() functions.
12. Refer to the binary search tree (BST) program C2.7 in Sect. 2.9.2.
(1) Prove that the insert() function does not insert nodes with the same key twice.
(2) Rewrite the main() function, which builds a BST by keys generated by rand() % 100, so that the keys are in the range of [0-99]
13. A shortcoming of the print tree by level program in Sect. 2.10.1.is that it does not show whether any child node is NULL. Modify the program to print a '–' if any child node is NULL.
14. For the Programming Project, implement a rename command, which changes the name of a node in the file tree.

```
rename(char *pathname, char *newname)
```

15. For the Programming project, design and implement a mv command, which moves pathname1 to pathname2

```
mv(char *pathname1, char *pathname2);
```

16. For the Programming Project, discuss whether we can use IN-ORDER or POST-ORDER traversal to save and reconstruct a binary tree.

References

Debugging with GDB, https://ftp.gnu.org/Manuals/gdb/html_node/gdb_toc.html, 2002
Linux vi and vim editor, http://www.yolinux.com/TUTORIALS/LinuxTutorialAdvanced_vi.html, 2017
GDB: The GND Project Debugger, www.gnu.org/software/gdb/, 2017
GNU Emacs, https://www.gnu.org/s/emacs, 2015
GNU make, https://www.gnu.org/software/make/manual/make.html, 2008
Wang, K. C., Design and Implementation of the MTX Operating system, Springer A.G, 2015
Youngdale, E. The ELF Object File Format: Introduction, Linux Journal, April, 1995