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. The Unix/Linux File System Tree

The logical organization of a Unix file system is a general tree, as shown in the figure 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, Figure 2.31 shows a binary tree which is equivalent to the general tree of Figure 2.28. In Figure 2.31, thin lines represent childPtr pointers and thick lines represent siblingPtr. For clarify, NULL pointers are not shown.

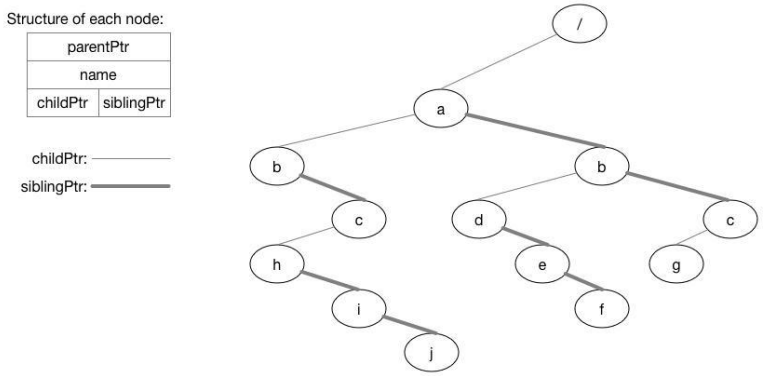


Figure 2.31. Implementation of General Tree by Binay Tree

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.

(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);
```

12.13.6. 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] …

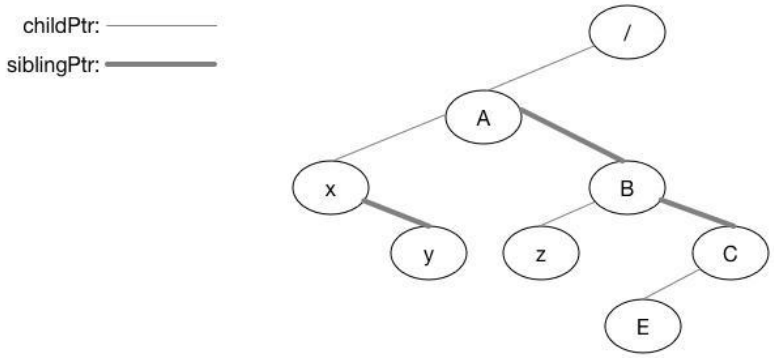
pwd

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



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

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

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 HELPs

(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 at
http://cs360.eecs.wsu.edu/~kcw/samples/ch2/project.bin
It's a binary executable file. The reader my download it and run it under Linux. Source code of the project solution is available to instructors upon request from the author.

Summary

Chapter 2 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 Section 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 Section 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 Section 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 Section 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 Section 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).

```
MPQ = level_1 : highest priority
      level_2 : 2nd highest priority
      ..........................
      level_n : lowest priority
```

(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 Section 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.