Workshop E

Operating Systems Programming – 300698
Operating Systems Programming (Advanced)– 300943

1 Introduction

In this workshop you will be implementing a file system simulator, based on the CP/M file system. A description of CP/M is given in section 2, and the tasks are detailed in the subsequent sections.

2 CP/M Information

The following is reproduced from Tanenbaum, A.S., (2001), *Modern Operating Systems*, 2nd, New York : Prentice-Hall pages 435–438.

6.4.2 The CP/M File System

The first personal computers (then called microcomputers) came out in the early 1980s. A popular early type used the 8-bit Intel 8080 CPU and had 4 KB of RAM and a single 8-inch floppy disk with a capacity of 180 KB. Later versions used the slightly fancier (but still 8-bit) Zilog Z80 CPU, had up to 64 KB of RAM, and had a whopping 720-KB floppy disk as the mass storage device. Despite the slow speed and small amount of RAM, nearly all of these machines ran a surprisingly powerful disk-based operating system, called **CP/M (Control Program for Microcomputers)** (Golden and Pechura, 1986). This system dominated its era as much as MS-DOS and later Windows dominated the IBM PC world. Two decades later, it has vanished without a trace (except for a small group of diehard fans), which gives reason to think that systems that now dominate the world may be essentially unknown when current babies become college students (Windows what?).

It is worth taking a look at CP/M for several reasons. First, it was a historically very important system and was the direct ancestor of MS-DOS. Second, current and future operating system designers who think that a computer needs 32 MB just to boot the operating system could probably learn a lot about simplicity from a system that ran quite well in 16 KB of RAM. Third, in the coming decades, embedded systems are going to be extremely widespread. Due to cost, space, weight, and power constraints, the operating systems used, for example, in watches, cameras, radios, and cellular telephones, are of necessity going to be lean and mean, not unlike CP/M. Of course, these systems do not have 8-inch floppy disks, but they may well have electronic disks using flash memory, and building a CP/M-like file system on such a device is straightforward.

The layout of CP/M in memory is shown in Fig. 6-30. At the top of main memory (in RAM) is the BIOS, which contains a basic library of 17 I/O calls used by CP/M (in this section we will describe CP/M 2.2, which was the standard version when CP/M was at the height of its popularity). These calls read and write the keyboard, screen, and floppy disk.

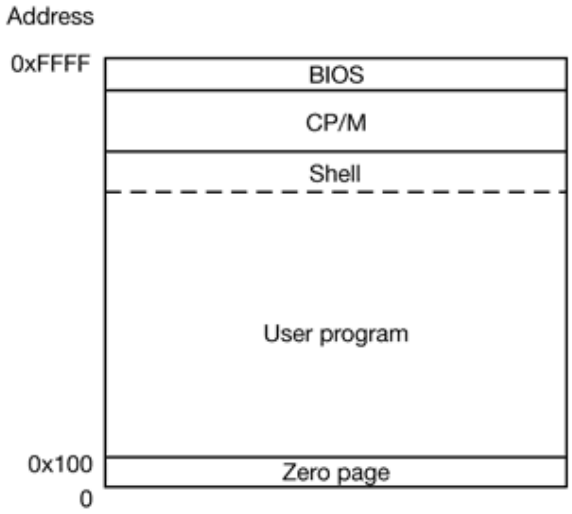

Figure 6-30. Memory layout of CP/M.

Just below the BIOS is the operating system proper. The size of the operating system in CP/M 2.2 is 3584 bytes. Amazing but true: a complete operating system in under 4 KB. Below the operating system comes the shell (command line processor), which chews up another 2 KB. The rest of memory is for user programs, except for the bottom 256 bytes, which are reserved for the hardware interrupt vectors, a few variables, and a buffer for the current command line so user programs can get at it.

The reason for splitting the BIOS from CP/M itself (even though both are in RAM) is portability. CP/M interacts with the hardware only by making BIOS calls. To port CP/M to a new machine, all that is necessary is to port the BIOS there. Once that has been done, CP/M itself can be installed without modification. A CP/M system has only one directory, which contains fixed-size (32-byte) entries. The directory size, although fixed for a given implementation, may be different in other implementations of CP/M. All files in the system are listed in this directory. After CP/M boots, it reads in the directory and computes a bitmap containing the free disk blocks by seeing which blocks are not in any file. This bitmap, which is only 23 bytes for a 180-KB disk, is kept in memory during execution. At system shutdown time it is discarded, that is, not written back to the disk. This approach eliminates the need for a disk consistency checker (like *fsck*) and saves 1 block on the disk (percentually equivalent to saving 90 MB on a modern 16-GB disk).

When the user types a command, the shell first copies the command to a buffer in the bottom 256 bytes of memory. Then it looks up the program to be executed and reads it into memory at address 256 (above the interrupt vectors), and jumps to it. The program then begins running. It discovers its arguments by looking in the command line buffer. The program is allowed to overwrite the shell if it needs the memory. When the program finishes, it makes a system call to CP/M telling it to reload the shell (if it was overwritten) and execute it. In a nutshell, that is pretty much the whole CP/M story.

In addition to loading programs, CP/M provides 38 system calls, mostly file services, for user programs. The most important of these are reading and writing files. Before a file can be read, it must be opened. When CP/M gets an open system call, it has to read in and search the one and only directory. The directory is not kept in memory all the time to save precious RAM. When CP/M finds the entry, it immediately has the disk block numbers, since they are stored right in the directory entry, as are all the attributes. The format, of a directory entry is given in Fig. 6-31.

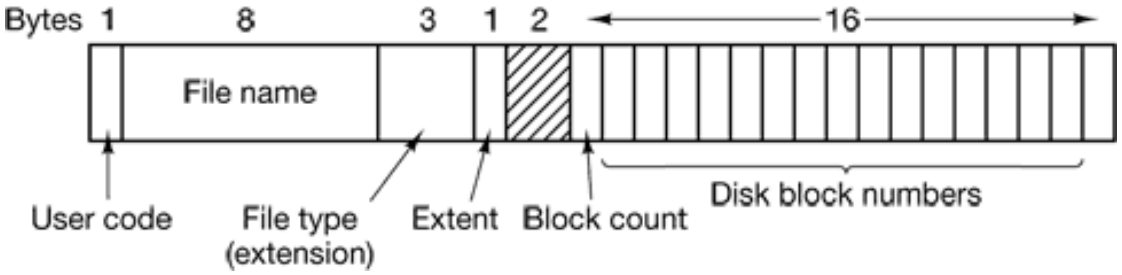

Figure 6-31. The CP/M directory entry format.

The fields in Fig. 6-31 have the following meanings. The *User code* field keeps track of which user owns the file. Although only one person can be logged into a CP/M at any given moment, the system supports multiple users who take turns using the system. While searching for a file name, only those entries belonging to the currently logged-in user are checked. In effect, each user has a virtual directory without the overhead of managing multiple directories.

The next two fields give the name and extension of the file. The base name is up to eight characters; an optional extension of up to three characters may be present. Only upper case letters, digits, and a small number of special characters are allowed in file names. This 8 + 3 naming using upper case only was later taken over by MS-DOS.

The *Block count* field tells how many bytes this file entry contains, measured in units of 128 bytes (because I/O is actually done in 128-byte physical sectors). The last 1-KB block may not be full, so the system has no way to determine the exact size of a file. It is up to the user to put in some END-OF-FILE marker if desired. The final 16 fields contain the disk block numbers themselves. Each block is 1 KB, so that maximum file size is 16 KB. Note that physical I/O is done in units of 128-byte sectors and sizes are kept track of in sectors, but file blocks are allocated in units of 1 KB (8 sectors at a time) to avoid making the directory entry too large.

However, the CP/M designer realized that some files, even on a 180-KB floppy disk, might exceed 16 KB, so an escape hatch was built around the 16-KB limit. A file that is between 16 KB and 32 KB uses not one directory entry, but two. The first entry holds the first 16 blocks; the second entry holds the next 16 blocks. Beyond 32 KB, a third directory entry is used, and so on. The *Extent* field keeps track of the order of the directory entries so the system knows which 16 KB comes first, which comes second, and so on.

After an open call, the addresses of all the disk blocks are known, making read straightforward. The write call is also simple. It just requires allocating a free block from the bitmap in memory and then writing the block. Consecutive blocks on a file are not placed in consecutive blocks on the disk because the 8080 cannot process an interrupt and start reading the next block on time. Instead, interleaving is used to allow several blocks to be read on a single rotation.

CP/M is clearly not the last word in advanced file systems, but it is simple, fast, and can be implemented by a competent programmer in less than a week. For many embedded applications, it may be all that is needed.

3 Simplifications

You are required to implement a simulation of a CP/M like file system, as described in the above section. The file system will be further modified and simplified in the following way:

- The disk size is 360 kbyte.
- The smallest unit of allocation is 4 kbyte.
- The maximum size of a file is 64 kbytes, so there is only one directory entry per file
- The main directory occupies the first block of the disk (block 0), and its size is fixed at 1 block, so there can only be 128 files in this file system.
- As the directory always occupies only the first block, therefore no control information about it needs to be stored in the directory (i.e. no `.` entry).
- the only user is user 1
- user -1 is not a valid user, and can be used to allocate free directory entries.

You are not supposed to implement the actual storage, only the control structures of the file system. When implementing the free bitmap you must use a bitmap, i.e. it should be an array, but each element of the array should represent several blocks.

4 Programming Tasks

Your task is to implement disk initialisation, including creating the root directory, and a free bitmap for this disk. You need to find a way to specify an empty directory entry. You may take advantage of the simplifications to the file system, which makes some of the fields in a CP/M directory entry unused. To test your implementation you need to provide the following functions, available from a menu:

Initialise Disk create and initialise disk control structures

List Files in the Directory the directory is initially empty

Display the Free Bitmap all blocks, except block 0, are initially free

Open/Create File if the file name specified is not in the directory, a new file will be created

Read File list the blocks occupied by the file (not the content of these blocks)

Write File allocate another block to the file, you should not preallocate blocks for the file, you should allocate the first available block

Delete File deallocate all blocks, and free the directory entry

You need to pay close attention to multiple boundary conditions, which exist in this file system, including the total size of the disk, maximum size of a file, maximum number of files etc.

5 Marking scheme

Following the marking scheme given in the Learning Guide, the following are the equally weighted functionality requirements.

- Menu appropriate
- directory structures appropriately initialised
- bitmap is appropriately initialised
- **List Files in the Directory** functions correctly
- **Print Bitmap** functions correctly
- **Open/Create File** functions correctly
- **Read File** functions correctly
- **Write File** functions correctly
- **Write File** obeys boundary conditions and allocation policy
- **Delete File** functions correctly
- **Delete File** updates bitmap correctly

Once the functionality score is determined it will be weighted by the following rubric, taken from the learning guide (with zero weighted criteria removed).

CRITERIA (Weighting)	Unsatisfactory (0%)	Poor (25%)	Good (50%)	Very good (75%)	Excellent (100%)
Readability (25%)	Code is unreadable.	The code is poorly organized and very difficult to read.	The code is readable only by someone who knows what it is supposed to be doing.	The code is fairly easy to read.	The code is exceptionally well organized and very easy to follow.
Documentation (25%)	No documentation provided.	The documentation is simply comments embedded in the code and does not help the reader understand the code.	The documentation is simply comments embedded in the code with some simple header comments separating routines.	The documentation consists of embedded comment and some simple header documentation that is somewhat useful in understanding the code.	The documentation is well written and clearly explains what the code is accomplishing and how.
Error/Exception Handling (50%)	Completed functional requirements handle no error/exception conditions.	Completed functional requirements handle obvious error/ exception conditions, or makes no distinction between benign and fatal errors/exceptions.	Completed functional requirements handle some error/exception conditions, most errors/exceptions correctly categorised as benign or fatal.	Completed functional requirements handle most error/exception conditions, most errors/exceptions correctly categorised as benign or fatal.	Completed functional requirements handle all error/exception conditions, with all errors/exceptions correctly categorised as benign or fatal.

6 File: cpm.h

```
#ifndef CPM_H
#define CMP_H
/* Prevent multiple inclusion */


/* cpm.h
   Various definitions for Workshop E
*/

/* The bitmap */
extern unsigned char bitmap[12];
/* 360Kb disk with 4Kb blocks -> 11.25 bytes for bitmap
   so round up to 12 */

/* The directory entry */
struct CPMdirent
{
  signed char usercode;
         char filename[9];
         char filetype[4];
         char extent;
         char blockcount;
         char blocks[16];
};
/* Modelled on the description in [1].

   The two unused bytes have been added to the
   end of the two name parameters to allow them
   to be C strings
*/

/* The Directory */
extern struct CPMdirent directory[128];

int toggle_bit(int block);
/* Toggles the value of the bit block, in
   the external array bitmap.
   returns the current value of the bit
   Does NOT validate block!!!
*/
int block_status(int block);
/* Returns if block block is allocated
   returns 0 if bitmap bit is 0, not
   0 if bitmap bit is 1
   Does NOT validate block!!!
*/
#endif
```

7 File: cpm.c

```
/* cpm.c
   Useful functions for Workshop E
*/
#include"cpm.h"
unsigned char bitmap[12];
struct CPMdirent directory[64];

int toggle_bit(int block)
{

        int elem=block/8;
        int pos=block%8;
        int mask=1<<pos;

        bitmap[elem]^=mask;

        return bitmap[elem]&mask;
}


int block_status(int block)
{
        int elem=block/8;
        int pos=block%8;
        int mask=1<<pos;

        return bitmap[elem]&mask;
}
```

References

[1] Tanenbaum, A.S., (2001), *Modern Operating Systems*, 2nd, New York : Prentice-Hall