

Operating Systems exercises – week 9 – Selected answers.

Tutorial questions

Longer questions.

1. Why do some systems keep track of the type of a file, while others leave it to the user or simply do not implement multiple file types? Which system is “better?”

Answer:

Some systems allow different file operations based on the type of the file (for instance, an ascii file can be read as a stream while a database file can be read via an index to a block). Other systems leave such interpretation of a file's data to the process and provide no help in accessing the data. The method which is “better” depends on the needs of the processes on the system, and the demands the users place on the operating system. If a system runs mostly database applications, it may be more efficient for the operating system to implement a database-type file and provide operations, rather than making each program implement the same thing (possibly in different ways). For general purpose systems it may be better to only implement basic file types to keep the operating system size smaller and allow maximum freedom to the processes on the system.

2. What are the advantages and disadvantages of recording the name of the creating program with the file's attributes (as is done in the Macintosh Operating System)?

Answer:

By recording the name of the creating program, the operating system is able to implement features (such as automatic program invocation when the file is accessed) based on this information. However, it does add overhead in the operating system and require space in the file descriptor.

3. Assume a system uses 2K blocks and 16 bit (2 byte) addresses. What is the largest file size that such a system can support using indexed block file allocation with a single level index. Comment on the resulting maximum size in terms of its adequacy for a modern filing system.

No answer – question and answer to be included in portfolio that is to be handed in.

4. Consider a file currently consisting of 100 blocks. Assume that the file control block (and the index block, in the case of indexed allocation) is already in memory. Calculate how many disk I/O operations are required for contiguous, linked, and indexed (single-level) allocation strategies, if, for one block, the following conditions hold. In the contiguous-allocation case, assume that there is no room to grow in the beginning, but there is room to grow in the end. Assume that the block information to be added is stored in memory.

- a. The block is added at the beginning.
- b. The block is added in the middle.
- c. The block is added at the end.
- d. The block is removed from the beginning.
- e. The block is removed from the middle.
- f. The block is removed from the end.

Answer:

	Contiguous	Linked	Indexed
a.	201	1	1
b.	101	52	1
c.	1	3	1
d.	198	1	0
e.	98	52	0
f.	0	100	0

Shorter questions.

4. What is a file?

Answer:

A named collection of related data defined by the creator, recorded on secondary storage.

5. What does OPEN do?

Answer:

$\text{open}(F_i)$ – search the directory structure on disk for entry F_i , and move the content of entry to open file table - returning a pointer that points to entry in open file table

6. What does CLOSE do?

Answer:

$\text{close}(F_i)$ – move the content of entry F_i in open file table to directory structure on disk.

7. What is a file path name?

Answer:

A list of the directories, subdirectories, and files we must traverse to reach a file from the root directory.

8. What problems might arise on deletion, if a file is shared?

No answer – question and answer to be included in portfolio that is to be handed in.

9. How can we solve this problem?

No answer – question and answer to be included in portfolio that is to be handed in.

10. Consider a system that supports the strategies of contiguous, linked, and indexed allocation. What criteria should be used in deciding which strategy is best utilized for a particular file?

Answer:

Contiguous – if file is usually accessed sequentially, if file is relatively small.

Linked – if file is large and usually accessed sequentially.

Indexed – if file is large and usually accessed randomly.

11. Explain first-fit, best-fit, and worst-fit methods of allocating space for contiguous files.

Answer:

First-fit: Scan available blocks of disk for successive free sectors; use the first area found that has sufficient space; do not scan beyond that point.

Best-fit: Search for smallest area large enough to place the file.

Worst-fit: Search for largest area in which to place the file.

12. What is external fragmentation in a system with contiguous files?

Answer:

The disk has files scattered all over; fragmentation occurs when there is enough empty space collectively for the next file, but there is no single gap large enough for the entire file to fit in.

13. Can linked allocation have external fragmentation? Internal fragmentation?

Answer:

External — no. Internal — Yes.

14. Can linked allocation be used for direct-access files?

Answer:

No

Windows exercises.

1) Start the Windows virtual machine [Start, then VMware, then VMware Player, and select Windows XP virtual machine]

Remember to press CTRL+G to direct input to the virtual machine and CTRL+ALT to direct it to the host computer.

2) We are going to do some more Windows system programming. We will be using the development environment we used before called Dev C/C++.

So start Dev C/C++, go to Start and you will find an icon for Dev C/C++ in the most recently used list of programs. Click on it.

3) This will bring up Dev C/C++. Close the Tip of the Day. You will now see the interface for Dev C/C++.

4) Click on File menu then New and then Source File. This will then bring up a text editor screen where previously it was greyed out. You can enter program code in that text area. Entry is normal and the editor responds to cursor keys and the mouse as you would expect.

Windows system programming.

5) We are going to write a Windows system program that searches through a directory to find and print out all the files that meet some given criteria. In order to do this we need some Windows system calls that we have not yet seen.

Directory searching

You will need include files like

```
#include <windows.h>
#include <stdio.h>
#include <string.h>
#include <io.h>
#include <tchar.h>
```

To search a directory for files that match some specific criteria and obtain file attributes, you require a search handle which is obtained by a call to FindFirstFile function. Files are then obtained using FindNextFile and the search terminated with FindClose.

FindFirstFile

```
HANDLE FindFirstFile (
LPCTSTR    lpFileName,
LPWIN32_FIND_DATA    lpffd)
```

Returns a search handle.

lpFileName – points to a null terminated string which has a pathname which can contain wildcard characters (* and ?)

lpffd – points to a WIN32_FIND_DATA structure that contains information about the first file or directory that satisfies the search criteria

WIN32_FIND_DATA structure is defined by windows as

DWORD	dwFileAttributes,
FILETIME	ftCreationTime,
FILETIME	ftLastAccessTime,
FILETIME	ftLastWriteTime,
DWORD	nFileSizeHigh,
DWORD	nFileSizeLow,
DWORD	dwReserved0,
DWORD	dwReserved1,
TCHAR	cFileName[MAX_PATH],
TCHAR	cAlternateFileName [14],

dwFileAttributes – contains flags of file attributes – particularly file and directory indicators – appropriate masks are defined, so FILE_ATTRIBUTE_DIRECTORY mask can be used to identify a file as a directory

ftCreationTime – what name says

ftLastAccessTime - what name says

ftLastWriteTime - what name says

nFileSizeHigh – high order 32 bits of file size (only used if file is very large)

nFileSizeLow – low order 32 bits of file size

dwReserved0

dwReserved1,

cFileName[MAX_PATH] – normal file name (note – not the pathname, just the file name)

cAlternateFileName [14] – DOS 8.3 file name

note – MAX_PATH is windows system defined as 260 chars long

So example in use

To find a set of files that match some given criteria in a directory such as all those with .txt file extension, you need to call FindFirstFile as

```
// need search string defined and WIN32_FIND_DATA structure variable declared  
// and a search handle variable
```

```
HANDLE searchHandle;  
LPCTSTR pathName = "*.txt";
```

```

WIN32_FIND_DATA findData;

// don't forget the &
// this gets info put into findData about first file that matches search criteria
if (!(searchHandle = FindFirstFile(pathName,&findData)))
    printf("No file found that matches: %s",pathName);

```

FindNextFile

FindFirstFile stores information about the search criteria used, which is then used by FindNextFile to get the next file in the directory that matches the search criteria.

```

BOOL FindNextFile (
HANDLE      hFindFile,
LPWIN32_FIND_DATA  lpffd)

```

Returns a BOOL FALSE if arguments are invalid or no more matching files are found

So example in use

```

// gets info about next file that matches criteria
FindNextFile (searchHandle, &findData);

```

FindClose

Once you have finished searching for files, use FindClose to close the search handle

```

BOOL FindClose (HANDLE hFindFile)

```

hFindFile – handle from FindFirstFile

Example in use is trivial,

```

FindClose(searchHandle);

```

6) So you need to put the above together to produce a program that searches the current directory for all files of type '*.exe'. For each file that your program finds you should output the filename. You can use the Windows program from Exercise week 6 or last week as a template for a Windows program. Most of the code will need to be deleted and replaced with code that uses the FindFirstFile, FindNextFile and FindClose system calls.

You can save the file in the Documents and Settings directory as we have done before. Before we run the program you need to make sure that there is a least one exe file in the directory. There will probably be the program you have just written, but you could try making some minor changes to the code, saving the code with a new name to give you several .exe files.

7) Once this program is working, you can try also searching subdirectories. To do this you need some more system calls as below.

Checking type of file

You can do this by

```
// assuming we have findData defined as before i.e. as
WIN32_FIND_DATA findData;
// then we can check using something like
```

```
if((findData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) == 0)
// it is a file or
if((findData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) != 0)
// it is a directory
```

Getting and setting current directory

Need current path defined

```
TCHAR Curr_Path [MAX_PATH + 1];
```

To get current directory we use

```
DWORD GetCurrentDirectory (
DWORD cchCurDir,
LPTSTR lpCurDir)
```

lpCurDir points to the character array that will hold the pathname of the directory after the call to GetCurrentDirectory. cchCurDir is the size of the character buffer.

It returns the size of the pathname put in the character array or if the pathname is too large for the character array then the actual size of the pathname i.e. the size of array that we need.

Example in use,

```
// gets into Curr_Path the pathname of the current working directory
```

```
GetCurrentDirectory(MAX_PATH+1, Curr_Path);
```

To set the current working directory we use,

```
BOOL SetCurrentDirectory (LPCTSTR lpPathName)
```

lpPathName is the name of the new current directory. It can be a relative pathname or a full pathname. It returns TRUE/FALSE depending upon success.

Note that because findData.cFileName only gives the name of the file or directory and not a path, you will need to use the standard string function strcat() to create a new directory pathname by concatenating current directory pathname with ‘\’ and then with the findData.cFileName. You will need a suitable path variable declared as,

```
LPTSTR filePath;
```

Then,

```
/* this set current working directory to that specified in file name in  
WIN32_FIND_DATA structure found by FindFirstFile or FindNextFile */
```

```
GetCurrentDirectory(MAX_PATH+1, Curr_Path); // get current directory path  
filePath = strcat(Curr_Path, "\\");          // add'\' to it  
filePath = strcat(filePath, findData.cFileName); // add new directory name  
SetCurrentDirectory(filePath);              // set the directory
```

8) Now modify the program from question 7) to locate a sub-directory and change the current working directory to the sub-directory and search for .exe files there as well. You can identify a sub-directory by checking the file type for a directory. Of course, remember that the current directory is known as “.” and the parent directory as “..” and these 2 directories will be found in the directory listing. So this means that you will need to check that the name of the directory you have found is not “.” or “..” as you don’t want to look at parent directories, but just sub-directories for .exe files.

To do this you will need,

```
if ( (lstrcmp(findData.cFileName, T(".")) != 0) && (lstrcmp(findData.cFileName, T("..")) != 0) )  
// then code for doing whatever you want in here
```

// ANSWER – to questions 6, 7, & 8

```
#include <windows.h>
#include <stdio.h>
#include <string.h>
#include <io.h>
#include <tchar.h>

int main(int argc, LPTSTR argv[])
{

    HANDLE searchHandle;
    WIN32_FIND_DATA findData;
    LPCTSTR pathName = "*";
    TCHAR Curr_Path[MAX_PATH+1];
    LPTSTR filePath;

    if (!(searchHandle = FindFirstFile(pathName,&findData))) {
        printf("No file found that matches: %s",pathName);
        return(1);
    }

    printf("First file: %s\n",findData.cFileName);

    while (FindNextFile(searchHandle, &findData)) {
        printf("file name = %s\n",findData.cFileName);
        if ((findData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) != 0){

            if ( (lstrcmp(findData.cFileName,_T(".")) != 0) &&
                (lstrcmp(findData.cFileName,_T("..")) != 0) ) {

                GetCurrentDirectory(MAX_PATH+1,Curr_Path);
                filePath = strcat(Curr_Path, "\\");
                filePath = strcat(filePath, findData.cFileName);
                printf("filepath = %s\n",filePath);
                SetCurrentDirectory(filePath);
                GetCurrentDirectory(MAX_PATH+1,Curr_Path);
                printf("path=%s\n",Curr_Path);
            }
        }
    }
}
```