CSI2131 FILE MANAGEMENT

Prof. Lucia Moura

Winter 2002

LECTURE 1: INTRODUCTION TO FILE MANAGEMENT

Contents of today's lecture:

- Introduction to file structures
- History of file structure design
- Course contents and organization

References :

- Folk, Zoellick and Riccardi, File Structures, 1998. Sections 1.1 and 1.2.
- Course description handout (for course contents and organization)

Introduction to File Structures

- **Data processing from a computer science perspective:**

  - Storage of data
  - Organization of data
  - Access to data
  - Processing of data

  This will be built on your knowledge of Data Structures.

- **Data Structures vs File Structures**

  Both involve :

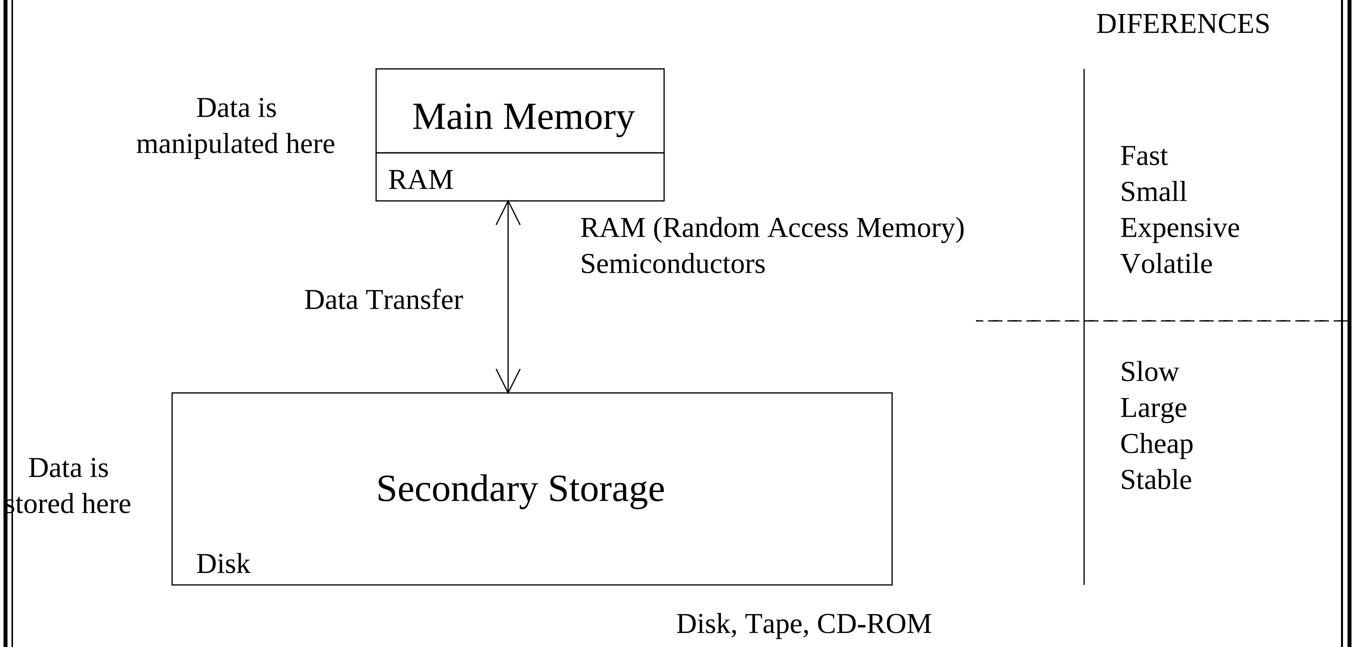
  Representation of Data
  +
  Operations for accessing data

  Difference :

  Data Structures : deal with data in main memory.

  File Structures : deal with data in secondary storage (Files)

Computer Architecture



How fast is main memory in comparison to secondary storage ?

Typical time for getting info from:

main memory: $\sim$ 12 nanoseconds $= 120 \times 10^{-9}$ secs
magnetics disks: $\sim$ 30 milliseconds $= 30 \times 10^{-3}$ secs

An analogy keeping same time proportion as above:

Looking at the index of a book: 20 secs
versus
Going to the library: 58 days

Main Memory

- Fast (since electronic)
- Small (since expensive)
- Volatile (information is lost when power failure occurs)

Secondary Storage

- Slow (since electronic and mechanical)
- Large (since cheap)
- Stable, persistent (information is preserved longer)

Goal of the file structure and what we will study in this course:

- Minimize number of trips to the disk in order to get desired information. Ideally get what we need in one disk access or get it with as few disk accesses as possible.
- Grouping related information so that we are likely to get everything we need with only one trip to the disk (e.g. name, address, phone number, account balance).

History of File Structure Design

1. In the beginning ... it was the tape
   - **Sequential access**
   - Access cost proportional to size of file
     [Analogy to sequential access to array data structure]

2. Disks became more common
   - **Direct access** [Analogy to access to position in array - binary search in sorted arrays]
   - **Indexes** were invented
     - list of keys and pointers stored in small file
     - allows direct access to a large primary file

     Great if index fits into main memory.
     As a file grows we have the same problem we had with a large primary file.

3. Tree structures emerged for main memory (1960's)
   - Binary search trees (BST's)
   - **Balanced**, self adjusting BST's : e.g. AVL trees (1963)

4. A tree structure suitable for files was invented : **B trees** (1979) and **B+ trees**

   Good for accessing millions of records with 3 or 4 disk accesses.

5. What about getting info with a single request ?

   - **Hashing Tables** (Theory developed over 60's and 70's but still a research topic)

     Good when files do not change to much in time.

   - **Extendible, dynamic hashing** (late 70's and 80's)

     One or two disk accesses even if file grows dramatically

Course Contents and Organization

- Introduction to file management. Fundamental file processing operations. (Chapters 1 and 2)
  Managing files of records. Sequential and direct access. (Chapters 4 and 5)
- Secondary storage, physical storage devices: disks, tapes and CD-ROM. (Chapter 3)
  System software: I/O system, file system, buffering. (Chapter 3)
- File compression: Huffman and Lempel-Ziv codes. Reclaiming space in files. Internal sorting, binary searching, keysorting. (Chapter 6)
- File Structures:
  - Indexing. (Chapter 7)
  - Co-sequential processing and external sorting. (Chapter 8)
  - Multilevel indexing and B trees. (Chapter 9)
  - Indexed sequential files and B+ trees. (Chapter 10)
  - Hashing. (Chapter 11)
  - Extendible hashing. (Chapter 12)

Chapters above refer to the textbook:
FOLK, ZOELLICK AND RICCARDI, File Structures, 1998.

Refer to the "course description handout" for course organization.

LECTURE 2: FUNDAMENTAL FILE PROCESSING OPERATIONS

Contents of today's lecture:

- Sample programs for file manipulation
- Physical files and logical files
- Opening and closing files
- Reading from files and writing into files
- How these operations are done in C and C++
- Standard input/output and redirection

References :

- FOLK, ZOELLICK AND RICCARDI, File Structures, 1998. Section 2

Sample programs for file manipulation

A program to display the contents of a file on the screen:

- Open file for input (reading)
- While there are characters to read from the input file :

  Read a character from the file

  Write the character to the screen
- Close the input file

A C program (which is also a valid C++ program) for doing this task:

```
// listc.cpp
#include <stdio.h>

main() {
    char ch;
    FILE * infile;

    infile = fopen("A.txt","r");

    while (fread(&ch,1,1,infile) != 0)
        fwrite(&ch,1,1,stdout);
    fclose(infile);
}
```

A C++ program for doing the same task:

```
// listcpp.cpp
#include <fstream.h>

main() {
    char ch;
    fstream infile;

    infile.open("A.txt",ios:in);
    infile.unsetf(ios::skipws);
         // set flag so it doesn't skip white space

    infile >> ch;
    while (! infile.fail())  {
        cout << ch ;
        infile >> ch ;
    }
    infile.close();
}
```

Physical Files and Logical Files

physical file: a collection of bytes stored on a disk or tape

logical file: a "channel" (like a telephone line) that connects the program to a physical file

- The program (application) sends (or receives) bytes to (from) a file through the logical file. The program knows nothing about where the bytes go (came from).

- The operating system is responsible for associating a logical file in a program to a physical file in disk or tape. Writing to or reading from a file in a program is done through the operating system.

Note that from the program point of view, input devices (keyboard) and output devices (console, printer, etc) are treated as files - places where bytes come from or are sent to.

There may be thousands of physical files on a disk, but a program only have about 20 logical files open at the same time.

The physical file has a name, for instance `myfile.txt`

The logical file has a logical name used for referring to the file inside the program. This logical name is a variable inside the program, for instance `outfile`

In C programming language, this variable is declared as follows:

```
FILE * outfile;
```

In C++ the logical name is the name of an object of the class `fstream`:

```
fstream outfile;
```

In both languages, the logical name `outfile` will be associated to the physical file `myfile.txt` at the time of **opening** the file as we will see next.

Opening Files

Opening a file makes it ready for use by the program.
Two options for opening a file :

- open an **existing** file
- create a **new** file

When we open a file we are positioned at the beginning of the file.

How to do it in C:

```
FILE * outfile;
outfile = fopen("myfile.txt", "w");
```

The first argument indicates the physical name of the file.
The second one determines the “mode”, i.e. the way, the file is opened.
The mode can be:

- `"r"`: open an existing file for input (reading);
- `"w"`: create a new file, or truncate existing one, for output;
- `"a"`: open a new file, or append an existing one, for ouput;
- `"r+"`: open an existing file for input and output;
- `"w+"`: create a new file, or truncate an existing one, for input and output;
- `"a+"`: create a new file, or append an existing one, for input and output.

How to do it in C++:

```
fstream outfile;
outfile.open("myfile.txt",ios::out);
```

The second argument is an integer indicating the mode.
Its value is set as a "bitwise or" (operator `|`) of constants defined in the class `ios`:

- `ios::in` open for input;
- `ios::out` open for output;
- `ios::app` seek to the end of file before each write;
- `ios::trunc` always create a new file;
- `ios::nocreate` fail if file doesn't exist;
- `ios::noreplace` create a new file, but fail if it already exists;
- `ios::binary` open in binary mode (rather than text mode).

Exercise: Open a physical file `"myfile.txt"` associating it to the logical file `"afile"` and with the following capabilities:

1. input and output (appending mode):
   `afile.open("myfile.txt", ios::in|ios::app);`
2. create a new file, or truncate existing one, for output:
3. open an existing file for input and output, no creation allowed:

Closing Files

This is like "hanging up" the line connected to a file.

After closing a file, the logical name is free to be associated to another physical file.

Closing a file used for output guarantees that everything has been written to the physical file.
We will see later that bytes are not sent directly to the physical file one by one; they are first stored in a buffer to be written later as a block of data. When the file is <u>closed</u> the leftover from the buffer is flushed to the file.

Files are usually closed automatically by the operating system at the end of program's execution.
It's better to close the file to prevent data loss in case the program does not terminate normally.

In C :

```
fclose(outfile);
```

In C++ :

```
outfile.close();
```

Reading

Read data from a file and place it in a variable inside the program.

A generic `Read` function (not specific to any programming language):

```
Read(Source_file, Destination_addr, Size)
```

`Source_file`: logical name of a file which has been opened
`Destination_addr`: first address of the memory block were data should be stored
`Size`: number of bytes to be read

In C (or in C++ using C streams):

```
char c;      // a character
char a[100]; // an array with 100 characters
FILE * infile;
  ⋮
infile = fopen("myfile,"r");
fread(&c,1,1,infile); /* reads one character */
fread(a,1,10,infile); /* reads 10 characters */
```

`fread`:

1st argument:	destination address (address of variable `c`)
2nd argument:	element size in bytes (a `char` occupies 1 byte)
3rd argument:	number of elements
4th argument:	logical file name

In C, read and write operations to files are supported by various functions: **fread, fget, fwrite, fput, fscanf, fprinf**.

In C++ :

```
char c;
char a[100];
fstream infile;
infile.open("myfile.txt",ios::in);
infile >> c;     // reads one character
infile.read(&c,1);
    // alternative way of reading one character
infile.read(a,10); // reads 10 bytes
```

Note that in the C++ version, the operator `>>` communicates the same info at a higher level. Since `c` is a char variable, it's implicit that only 1 byte is to be transferred.

C++ fstream also provide the `read` method, corresponding to `fread` in C.

Writing

Write data from a variable inside the program into the file.
A generic `Write` function :

```
Write (Destination_File, Source_addr, Size)
```

`Destination_file`: logical file name of a file which has been opened
`Source_addr`: first address of the memory block where data is stored
`Size`: number of bytes to be written

In C (or in C++ using C streams) :

```
char c; char a[100];
FILE * outfile;
outfile = fopen("mynew.txt","w");
/* omitted initialization of c and a */
fwrite(&c,1,1,outfile);
fwrite(a,1,10,outfile);
```

In C++ :

```
char c; char a[100];
fstream outfile;
outfile.open("mynew.txt",ios::out);
/* omitted initialization of c and a */
outfile << c;
outfile.write(&c,1);
outfile.write(a,10);
```

Detecting End-of-File

When we try to read and the file has ended, the read was unsuccessful. We can test whether this happened in the following ways :

In C : Check whether `fread` returned value 0.

```
int i;
i = fread(&c,1,1,infile); // attempted to read
if (i==0) // true if file has ended
    ...
```

in C++: Check whether `infile.fail()` returns `true`.

```
infile >> c; // attempted to read
if (infile.fail()) // true if file has ended
    ...
```

Alternatively, check whether `infile.eof()` returns `true`.
Note that `fail` indicates that an operation has been unsuccessful, so it is more general than just checking for end of file.

Logical file names associated to standard I/O devices and re-direction

purpose	**default meaning**	**logical name in C**	**logical name in C++**
Standard Output	Console/Screen	`stdout`	`cout`
Standard Input	Keyboard	`stdin`	`cin`
Standard Error	Console/Screen	`stderr`	`cerr`

These streams don't need to be open or closed in the program.

Note that some operating systems allow this default meanings to be changed via a mechanism called **redirection**.

In UNIX and DOS :

Suppose that `prog` is the executable program.
Input redirection (standard input becomes file `in.txt`):

```
prog < in.txt
```

Output redirection (standard output becomes file `out.txt`. Note that standard error remains being console):

```
prog > out.txt
```

You can also do:

```
prog < in.txt > out.txt
```