Data and File Structures

Unit – 4

(File Structures)

by
Dr. Sunil Pratap Singh
(Assistant Professor, BVICAM, New Delhi)

2021

 U4.1

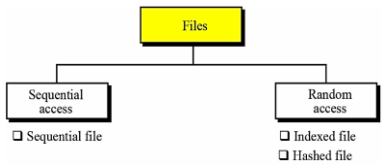
File

- A **file** is a collection of data stored on mass storage (e.g., disk or tape).
  - The data is subdivided into **records** (e.g., student information).
  - Each record contains a number of **fields** (e.g., roll number, name).
  - One (or more) field is the **key** field (e.g., roll number).

 U4.2

File Organizations

- A **file organization** refers to the way records are arranged on a storage device.
- How best the files be arranged for easy of access?



 U4.3

Sequential Files

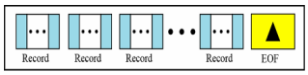
- A sequential file is one in which records can only be accessed one after another from beginning to end.
- This file organization is the simplest way to store and retrieve records of a file.
- In this file, data records are stored in some specific sequence, e.g., order of arrival, value of key field, etc.

Record 1			Record 2			Record 3		
001	Abhijeet	22	002	Arpit	21	003	Ashutosh	21
Key			Key			Key		

 U4.4

Sequential Files

- The records of a sequential file cannot be accessed at random, i.e., to access the nth record, one must traverse the preceding (n-1) records.



 U4.5

Sequential File Organization

- In sequential file organization, the actual storage of records might or might not be sequential:
  - On a tape, it usually is.
  - On a disk, it might be distributed across sectors and the operating system would use a linked list of sectors to provide the illusion of sequentially.

 U4.6

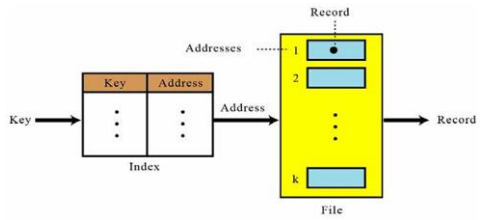
Sequential File Organization

- **Advantages:**
  - Easy to handle
  - Involve no overhead
  - Can be stored on tapes as well as disks
- **Disadvantages:**
  - Records can only be accessed in sequence
  - Time consuming

 U4.7

Indexed File

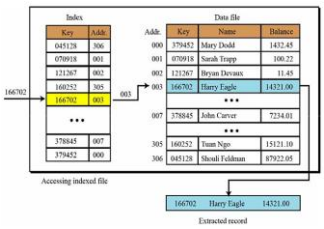
- To access a record in a file randomly, we need to know the address of the record.



 U4.8

Indexed File: Logical View

- An **indexed file** is made of a **data file**, which is a sequential file, and an **index**.
- The index itself is a **very small file** with only two fields: **the key of the sequential file** and **the address of the corresponding record on the disk**.
- The index is sorted based on the key values of the data files.



 U4.9

Indexed File: Accessing a Record

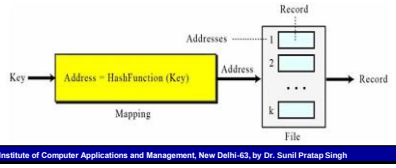
- Accessing a record in the file requires following steps:
  - The entire index file is loaded into main memory (the file is small and uses little memory).
  - The index entries are searched, using an efficient search algorithm such as a binary search, to find the desired key.
  - The address of the record is retrieved.
  - Using the address, the data record is retrieved and passed to the user.

Inverted File

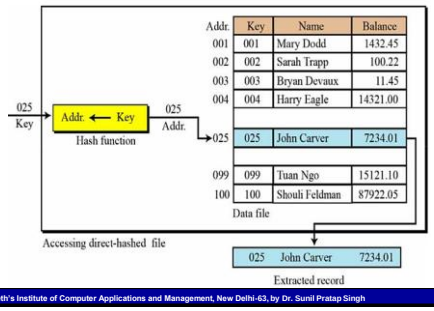
- One of the advantages of indexed files is that we can have **more than one index**, each with a different key.
- This type of indexed file is usually called an **inverted file**.

Direct File (Hashed File)

- A **hashed file** uses a mathematical function to map the key to the address.
- The user gives the key, the function maps the key to the address and passes it to the operating system, and the record is retrieved.



Process of Working in Direct File (Hashed File)



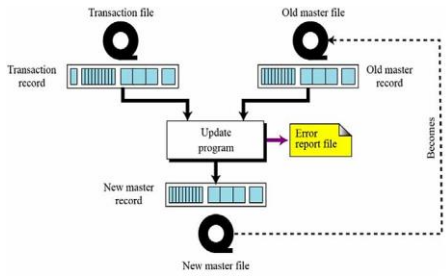
 U4.13

Updating a Sequential File

- Information that is kept on files needs to be modified as changes to the information on the file occur.
  - This process is called **UPDATING** and the files that are being update are usually called **MASTER FILES**.
- Updating a file can involve **ADDING, CHANGING or DELETING** records to/from the file.

 U4.14

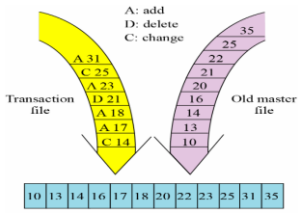
Process of Updating a Sequential File



 U4.15

Example of Updating a Sequential File

- To make the updating process efficient, all files are sorted on the same key.



Error in Updating a Sequential File

- Several cases may **create an error** and be reported in the error file:
  - If the transaction defines **adding a record that already exists** in the old master file (same key values).
  - If the transaction defines deleting or **changing a record that does not exist** in the old master file.

File Operations in C

- There are two distinct ways to perform file operations in C:
  - Low-level I/O Operations (uses UNIX system calls)
  - High-level I/O Operation (uses functions of C's standard I/O library)
- Data can be stored into files in two ways:
  - Text Mode
  - Binary Mode

Text Mode

- In text mode, data is stored as a line of characters where each character occupies 1 byte.
  - To store 123456 in a text file would take 6 bytes, 1 byte for each character.

1st byte	2nd byte	3rd byte	4th byte	5th byte	6th byte
0011 0001	0011 0010	0011 0011	0011 0100	0011 0101	0011 0110
'1'(49)	'2'(50)	'3'(51)	'4'(52)	'5'(53)	'6'(54)

This is how 123456 is stored in the file in text mode

  - In the text mode, what gets stored in the memory is that binary equivalent of the ASCII number of the character.

 U4.19

Binary Mode

- In binary mode, data is stored on a disk in the same way as it is represented in computer memory.
  - Storing 123456 in a binary mode would take only 2 bytes.

1110 0010	0100 0000

This is how 123456 is stored in the file in binary mode

  - Hence by using binary mode, we can save a lot of disk space.

 U4.20

High Level I/O Functions in 'C'

- `fopen()`: Opens an existing/creates a new file for use.
- `fclose()`: Closes a file which has been opened for use.
- `fscanf()`: Reads a set of data values from a file.
- `fprintf()`: Writes a set of data values to a file.
- `getc()`: Reads a character from a file.
- `putc()`: Writes a character to a file.
- `getw()`: Reads an integer from a file.
- `putw()`: Writes an integer to a file.
- `fseek()`: Sets the position to a desired point in the file.
- `rewind()`: Sets the position to the beginning of the file.

 U4.21

Operations on a File in 'C'

```
FILE *fp;
fp = fopen("fileName", "mode");
```

- Mode specifies the purpose of opening of the file. It can be one of the following:
  - `r` opens the file for reading only
  - `r+` opens the existing file for both reading and writing. If file does not exist, NULL is returned.
  - `w` opens the file for writing only
  - `w+` opens the file for both writing and reading. If the file exist, the previous contents are overwritten by new one.
  - `a` opens the file for appending (or adding) data to file.
  - `a+` opens the file for reading and appending. If the file does not exist, a new file is created.

Operations on a File in 'C' (contd…)

- When trying to open a file, one of the following things may happen:
  - When the mode is 'writing', a file with the specified name is created if the file does not exist. The contents are deleted, if the file already exists.
  - When the purpose is 'appending', the file is opened with the current contents safe. A file with the specified name is created if the file does not exist.
  - If the purpose is 'reading', and if it file exists, then it is opened with the current contents safe otherwise an error occurs.

Input/Output Operations on a File

- Using `getc()` and `putc()`

```
putc(ch, fp);
ch = getc(fp);
```

  - `putc(ch, fp);` is used to write the character contained in the character variable **ch** to the file associated with **FILE** pointer **fp**.
  - `c = getc(fp);` is used to read a character from a file that has been opened in read mode.

Input/Output Operations on a File (contd…)

- Using `getw()` and `putw()`

```
putw(in, fp);
in = getw(fp);
```

  - `getw()` and `putw()` are integer-oriented functions.
  - These are similar to the `getc()` and `putc()` functions and are used to read and write integer values.
  - `putw(in, fp);` is used to write the integer contained in the character variable **in** to the file associated with **FILE** pointer **fp**.
  - `in = getw(fp);` is used to read a character from a file that has been opened in read mode.

 U4.25

Input/Output Operations on a File (contd…)

- Using `fprintf()` and `fscanf()`

```
fprintf(fp, "control string", list);
fscanf(fp, "control string", list);
```

  - `fprintf()` and `fscanf()` are identical to the printf() and scanf() functions, except of course that they work on files.
  - `fprintf()` and `fscanf()` can handle a group of mixed data simultaneously.

 U4.26

Input/Output Operations on a File

- Using `getc()` and `putc()`

```
putc(ch, fp);
ch = getc(fp);
```

- Using `getw()` and `putw()`

```
putw(in, fp);
in = getw(fp);
```

- Using `fprintf()` and `fscanf()`

```
fprintf(fp, "control string", list);
Example: fprintf(fp, "%s %d %f", item, number, price);
When the end of line is reached, fscanf() returns EOF.
```

 U4.27

Errors during I/O Operations

- During I/O operations, an error may occur due to the following reasons:
  - Trying to read beyond the end-of-file mark.
  - Trying to use a file that has not been opened.
  - Trying to perform an operation on a file, when the file is opened for another type of operation.
  - Opening a file with an invalid file name.
  - Attempting to write to a write-protected file.

Error Handling during I/O Operations

- `feof()` function can be used to test for an end of file condition.
  - It takes a FILE pointer as its only argument and returns a nonzero integer value if all of the data from the specified file has been read, and returns zero otherwise.

```
if(feof(fp) != 0)
  printf("End of data.\n");
```

- `ferror()` function reports the status of the file indicated.
  - It takes a FILE pointer as its argument and returns a nonzero integer if an error has been detected up to that point, during processing. It returns zero otherwise.

```
if(ferror(fp) != 0)
  printf("An error has occurred.\n");
```


Error Handling during I/O Operations (contd…)

- Whenever a file is opened using `fopen()` function, a file pointer is returned.
  - If the file cannot be opened for some reason, then the function returns a NULL pointer.
  - This facility can be used to test whether a file has been opened or not.

```
if(fp == NULL)
  printf("File could not be opened.\n");
```

Random Access to Files

- `ftell()` takes a file pointer and return a number of type long, that corresponds to the current position.
- This function is useful in saving the current position of a file, which can be used later in the program.
- It takes the following form: `n = ftell(fp);`
  - n would give the relative offset (in bytes) of the current position, which means that n bytes have already been read (or written).

 U4.31

Random Access to Files (contd…)

- `rewind()` takes a file pointer and resets the position to the start of the file.
  - The statement `rewind(fp); n = ftell(fp);` will assign **0** to **n** because the file position has been set to the start of the file by **rewind**.
- This function helps us in reading a file more than once, without having to close and open the file.
  - Whenever a file is opened for reading or writing, a rewind is done implicitly.
- Note: The first byte in the file is numbered as 0, second as 1, and so on.

 U4.32

Random Access to Files (contd…)

- `fseek()` function is used to move the file position to a desired location within the file.
- It takes the following form: fseek(fileptr, offset, position);
  - **fileptr** is a pointer to the file concerned; **offset** is a number or variable of type long; **position** is an integer number.
  - The offset specifies the number of positions (bytes) to be moved from the location specified by position.
  - The position can take one of the following three values: 0 (beginning of file), 1 (current position) and 3 (end of file).
  - The offset may be positive (move forwards) or negative (move backwards).

 U4.33

Examples of Operations of the fseek()

Statement	Meaning
fseek(fp,0L,0);	Go to the beginning. (Similar to rewind)
fseek(fp,0L,1);	Stay at the current position. (Rarely used)
fseek(fp,0L,2);	Go to the end of the file, past the last character of the file.
fseek(fp,m,0);	Move to (m+1)th byte in the file.
fseek(fp,m,1);	Go forward by m bytes.
fseek(fp,-m,1);	Go backward by m bytes from the current position.
fseek(fp,-m,2);	Go backward by m bytes from the end. (Positions the file to the mth character from the end.)

 U4.34

Input/Output Operations in Binary File

- `fread()` and `fwrite()` functions are commonly used to read and write binary data to and from the file respectively.
- `fwrite(void *ptr, int size, int n, FILE *fp);`
- `fread(void *ptr, int size, int n, FILE *fp);`
  - `ptr` points to the block of memory which contains the data items to be written.
  - `size` specifies the number of bytes of each item to be written.
  - `n` is the number of items to be written.
  - `fp` is a pointer to the file where data items will be written.
  - **On success, these functions returns the number of items successfully written/read to/from the file.**

 U4.35

Thank You

 U4.36