CSE 5031 Operating Systems
2020/21 Fall Term

Project: 2 – Part 2
Topic: Programming ISAM with Low-level I/O API
Date: 02 - 09.11.2020

Objectives:

- Implementing Indexed Sequential File Access Method
- Using low-level I/O **GNU C Library API**

References:

- **The GNU C Library Reference Manual** (http://www.gnu.org/software/libc/manual/pdf/libc.pdf)
- **Linux System Programming 2d ed., Robert Love, O'Reilly 2013** (http://pdf-ebooks-for-free.blogspot.com.tr/2015/01/oreilly-linux-system-programming.html)

Section A. Linux File Concept and I/O APIs

A.1 UNIX/Linux File Concept

UNIX abstracted the **files** stored on any magnetic storage as an **array of bytes** - characters-. **UNIX** extended **file** abstraction over the years to cover:

- ✓ data sets stored on **non-magnetic medium** (various, I/O devices, virtual terminals, inter-process communication channels); and
- ✓ almost all **system entities** that generate or store data, for instance (directories, processes, memory etc.).

To avoid confusions **UNIX/Linux** refers to **files** stored on any magnetic storage as **ordinary files**.

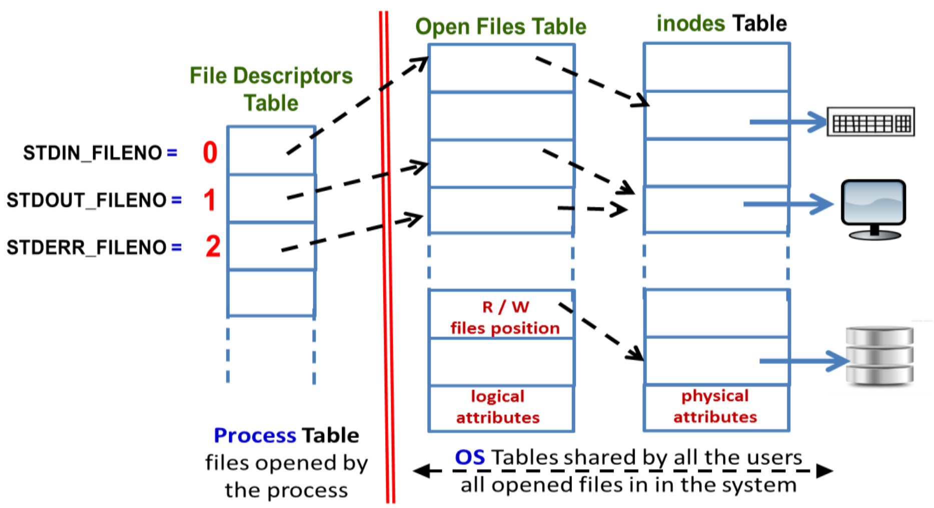
A.2 GNU UNIX/Linux File Access API

GNU C library provides two APIs to handle **Linux** files:

- ✓ low level I/O interface; and
- ✓ stream I/O interface**.**

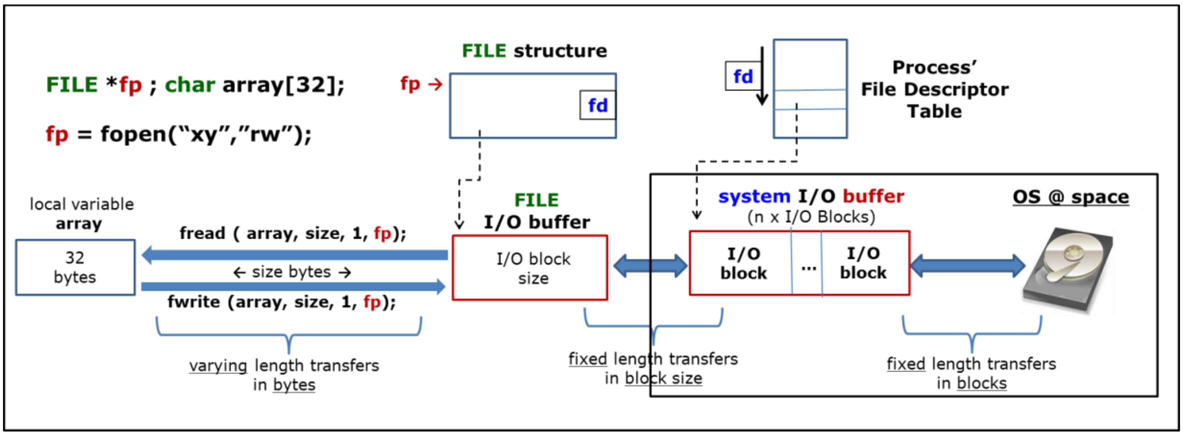
These APIs represent the **connection** between a **C** program and **the file entity** using respectively:

- ✓ **File descriptors**, **integer ordinals** e.g. **0,1,2, ..n** referring to an open file in the **File Descriptor Table**, that is stored in the user **virtual address space** in the **OS Kernel zone**; and
- ✓ **FILE Pointers**, referring to a **streams** data structure that is located in the in the user **virtual address space;** that provide a higher-level interface, layered on top of the **low-level I/O** mechanism and refer to it using **file descriptor** as shown on the figure at the next page.



Both **API** can also represent a connection to a device (such as a terminal), or a pipe or socket for communicating with another process, as well as to logical system entities such as process and file descriptor tables. But,

- ✓ to perform **control operations** that are specific to a particular kind of device, **low-level I/O API** should be used; it provides only simple functions for transferring **blocks of characters;**
- ✓ the main advantage of using the **stream I/O** interface is its richer and more powerful **I/O functions** set e.g. powerful formatted input & output functions (printf and scanf) as well as functions for character- and line-oriented input & output.
- ✓ **stream I/O** interface has **better performance** in transferring large number of **very small** size I/O data chunks versus low l **low-level I/O API** (try to explain why?).



A.3 Duality of Low-level and Stream I/O APIs

Since **streams** are implemented over **low-level I/O** mechanism, you can extract the **file descriptor** from a **stream** by calling the "**fileno**" function and perform **low-level operations** directly on the file descriptor. You can also initially open a connection as a **file descriptor** by calling the `**fdopen**" function and then make a **stream** associated with that file descriptor (refer to **GNU C Library Reference Manual** section 13.4).

In general, you should stick with using only **one I/O API**, unless there is some specific operation you want to do with the other. If you are concerned about portability of your programs to systems other than GNU, low-level I/O API be aware that file descriptors **are not as portable** as streams. You can expect any system running **ISO C** to support **streams**, but **non-GNU** systems may not support **low-level IO API** at all or may only implement a subset of it.

A.4 File Position

File position is a **pointer** to the current byte to be read from or written to a file in the byte array abstraction It is an attribute of an **opened file**. On **GNU** systems, **file position** is represented with an integer which counts the byte **offset** – *the number of bytes-* from the beginning of the file.

When a file is opened for reading or writing, its **file position** is set to the **beginning** of the file, at the byte **offset 0**. Each time a byte is read or written the file position is incremented. In other words, file **access mode** is **sequential**.

Alternatively, an existing file may be opened with the "**append**" attribute to add new records at the end of the file. Such an **open** sets the **file position** to the **end of the file**; if the file size is "n" bytes, the offset is set to "n" since file offset count starts at 0.

A.5 Random/Direct Access Modes

Ordinary files (data files in UNIX/Linux terminology) permit read or write operations **at any position** within the file. The **file position** may be set:

- ✓ when the file is opened; and successive reads or writes increments the **file position** by the amount of data transferred is or out;
- ✓ to any location using the **fseek** function on a stream (*Section 12.18 File Positioning*), or the **lseek** function on a file descriptor (*Section 13.2 Input and Output Primitives*); read / write operations proceed **sequentially** from the **new position**.

This type of **file access mode** is called is **sequential** whereas the second **random** or **direct access**.

Section B. Indexed File Organization

B.1 Random / Direct File Access the Rationale

Searching a record in a file **sequentially** yields in poor performance when the file contains a **large number** of records. The program must read all the records **one by one** until targeted record is reached, i.e. one or several of its fields are identified by the record search criteria.

Given a file hat contains "**n**" records, a sequential search for a given record requires the reading of "**n/2**" records **in average**. The process yields in **long access times** and generates a **large number of I/O** operations.

The alternative involves:

- ✓ the acquisition of the record's **file position** (i.e. by looking up in a list of "record key-file position" pairs);
- ✓ setting the **file position** to the position of targeted record (an I/O operation that does not involve a file access but just an update of its data structure in main memory); and
- ✓ reading/writing the record at the new offset (1 read/write).

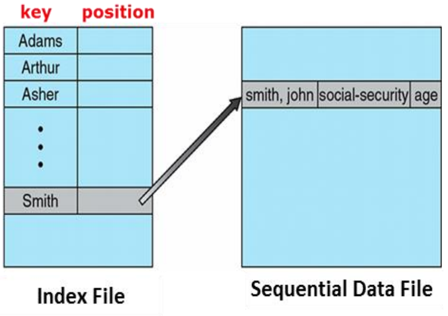
The performance gain achieved by **direct/random** access method is **1 access** versus **n/2 operations**.

B.2 Indexed File Organization

An effective **direct/random** file access method that enhances the search performance to sequential files is **ISAM**, the *Indexed Sequential Access Method*. **ISAM** uses an **index table** that matches the **key field** of a record with its **file position**. The figure here after depicts a simplified **ISAM** implementation that maps the **family name** of a given customer to its **file position**.

To retrieve a given record using **ISAM** a search program should:

- ✓ acquire the **key** of the record to access;
- ✓ search the **Index Table** in memory for the record **key**;
  if successful:
  - o set **file position** to the corresponding record offset extracted from the table using "**lseek**" or "**fseek**" primitive;
  - o **access** the record if positioning is successful.



ISAM requires:

- ✓ the generation of a sorted **index table** associated with the file;
- ✓ reading of the entire "**Index File**" in memory, in the **Index Table**, prior to any access;
- ✓ sorting eventually the data file in the primary key order, in case the **record key is not unique** e.g. several records with the same family name exist; in such case the **Index Table** will be used to access the first record with the same record key and the rest will be browsed in sequence.

B.3 Creating an ISAM File

In this project you will implement a the **ISAM** file organization for the ***"/etc/passwd"*** file which contains a **unique** record for **each account.** You will build the **index file** using the "***prj2-mkidx.c***", that:

- ✓ reads the "***passwd***" file, the copy of the ***"/etc/passwd"*** file stored in the current working directory;
- ✓ creates it's **Index Table**, using the first field (the **account name**) as the record key;
- ✓ sorts the **Index Table** in **ascending order** of the record key field;
- ✓ stores the **Index Table** as the "***passwd.idx***" file in the current working directory.

The "***prj2-mkidx.c***" program is stored at the course web site in the folder "**Resources/RefCprograms**" under "**CSE5031-Home**" tab. It uses:

- ✓ **stream I/O API** to read varying length ***"./passwd"*** file records; and
- ✓ **low-level I/O API** to create "***passwd.idx***" file with proper file attributes (ownership and access rights).

Analyze "***prj2-mkidx.c***" taking into consideration the following implementation details:

- ✓ ***"passwd"*** file is a byte array, containing a **variable length** record per account. Each record consists of ***7*** varying length fields separated by the "**:**" character and terminated by the new line ("***\n***") marker.

root:x:0:0:root:/root:/bin/bash\nbin:x:1:1:bin:/bin:/sbin/nologin\ndaemon:x:......

↑offset **0** ↑offset **32** ↑offset **64**

Note that, the record of the account **root** starts at the byte **position 0**; the record of **bin** is at **offset 32**; and **daemon** account starts at the offset 65 etc.

- ✓ As the fields of ***"/etc/passwd"*** are variable length, the following assumptions are made on the **max.** lengths:
  - max. lengths for the file name (**NAME_MAX**), and the path (**PATH_MAX**) are defined in the system file ***"/usr/include/linux/limits.h"***; but they are not used in "***prj2-mkidx.c***";
  - max. length of the user/account name is defined as **32 byte** in the manual for the "**useradd**" command;
  - max. record length of ***"/etc/passwd"*** is assumed to be less than **1023** bytes.

B.4 Creating Project's ISAM File Index

Perform the following to create the **index file** "**passwd.idx**":

- ✓ logon as "**sysadmin**";
- ✓ create the folder "***prj2***" in your home directory; and set it as your working directory;
- ✓ copy the ***"/etc/passwd"*** file;
- ✓ copy "***prj2-mkidx.c***" file posted at the course web site;
- ✓ make sure that the owner of "***prj2-mkidx.c***" file is "sysadmin", if not change it using the "**chown**" command;
- ✓ make sure that "***prj2-mkidx.c***" file has **read** and **write** access rights set for the owner; if not change them using the "**chmod**" command;
- ✓ compile "***prj2-mkidx.c***" with gcc, and name it "**mkidx**";
- ✓ run "**mkidx**" and enter the number of records **"passwd"** contains;
- ✓ make sure that "**passwd.idx**" has been created with correct content and has proper access control rights.

Section C. Developing the ISAM Query Program

C.1 Query Program Accessing "passwd" with ISAM

Write a query program in **C** that retrieves a series of selected records from the ***"passwd"*** file with the **ISAM** method using the **Index File** "**passwd.idx"** you have created in section **B.4**.

The query program should:

- ✓ load the file index "**passwd.idx**" in a dynamically allocated **Index Table** (note that you may derive the size of the Index Table from the size of the Index File using the "**stat**" function);
- ✓ read from **standard input** an account name, until an **end of file** is entered (ctrl+del keystrokes for the VM);
- ✓ retrieve the corresponding record from using **ISAM** method;
- ✓ display the **home directory** and **login shell** of the account retrieved if the search is successful.

The query program:

- ✓ **must be written** using low level **I/O API** to read the index "**passwd.idx**"; and to access the **"passwd"** file;
- ✓ may use **Stream I/O API** for **"stdin"** and "**stdout**" files.

✓ **Chapter 2** "**FILE I/O**" of the "**Linux System Programming"** cited under **References** contains authoritative **C** programming examples with the **low-level I/O API**. You are strongly advised to refer to this programming resource, instead of wasting your valuable time in "**fishing junk**" over the Internet.

C.2 Project Report

i) Run your query program with several accounts and store the results in an output text file.

ii) If your **program** is **operational,**

- add a comment line consisting of your name and student-id**;**
- store its **source code** and the **results** files in the "**Prj2-Part1**" folder, located at the course web site under the tab **CSE5031 - OS Section -X/Assignment**; where "**X**" stands for (1,2,3,4) the laboratory session group you are registered in.

Warning

You are encouraged to discuss the implementation procedures and general concepts behind the projects with your fellow students. However, **plagiarism is strictly forbidden**! Submitted report should be the result of **your personal work**!

Be advised that you are **accountable** of your submission not only for this project, but also for the mid-term, and final examinations. Your project grade may be reevaluated retrospectively, had you fail to answer correctly the same or a similar examination questions that you have solved with success in your submissions.