Creating Data structures on persistent storage:

1. File opening:

We open the file using the function fopen which returns a pointer to structure FILE.

We use this pointer in all file operations. If the opening fails, the function fopen returns NULL.

We should always check this return value before using the file pointer.

There are different modes in which thefile can be opened.

Check the man page of fopen function.

FILE \*fopen(const char \*pathname, const char \*mode);

Modes :

1. “r” : read mode; The file shall be opened for reading. It fails if the file does not exist.

There are other reasons as well for failure. We shall not discuss these anymore.

2. “w” : write mode: The file shall be opened for writing. If the file does not exist, it is created. If the file exists, it shall be truncated.

3. “r+” read/write mode: The file should exist. Otherwise, fopen fails and returns NULL. The file is opened for reading as well as writing. We can perform any of these operations.

4. “w+” read/write mode: If the file does not exist, it is created. If the file exists, it is truncated. We can then perform read as well as write.

There are other modes we shall not discuss in this document.

Check the program ex1.c.

In this case, the file opening fails if there is no file called test.dat.

// ex1.c

FILE\* fp;

// cannot create a file using r+

fp = fopen("test.dat", "r+");

if(fp == NULL)

{

printf("opening file failed\n");

}

else

{

printf("opening file succeeded\n");

fclose(fp);

}

Observe that fclose should be performed only if fopen is successful.

We want to create a data structure on secondary storage which should persist. We may create the file today, put some data and then come back later to access it – for both reading and writing.

We cannot use w+ as the mode. The file would be truncated in this mode.

We should use r+ as the mode. This would not work first time we try to open the file.

The program ex2.c gives a workaround to open the file in r+ mode. We open in r+ mode and if that fails, we open in w+ mode. We are assuming that the former fails as the file does not exist.

// ex2.c

FILE\* fp;

// cannot create a file using r+

fp = fopen("test.dat", "r+");

if(fp == NULL)

{

printf("opening file failed for r+\n");

fp = fopen("test.dat", "w+");

if(fp == NULL)

{

printf("opening failed for w\n");

}

else

{

printf("opening file succeeded for w\n");

}

}

2. We are creating a file of structures on the harddisk. Observe all the files in the dir3.

Let us took at each file.

The interface file **person3.h** defines the type person\_t.

struct person

{

char name[20];

int age;

};

typedef struct person person\_t;

extern size\_t size;

The size of the structure shall be required while writing to as well as reading from the file. We will cache the value of sizeof this structure in the global variable size.

The implementation file **person3.c** as of now has no implementation but for finding the size.

#include <stdio.h>

#include <stdlib.h>

#include "person3.h"

size\_t size = sizeof(person\_t);

All the file operations are supported by the interface file **file3.h.**

The signatures of these functions are very clear.

FILE\* open\_file(const char\* fname); // opens the file whose name is passed as argument.

// It opens the file r+ mode if the file already exists or w+ mode if the file does not exist.

void close\_file(FILE \*fp);

This closes the file successfully opened by open\_file.

void write\_file(FILE \*fp, const person\_t \*ptr\_person);

// This dumps to the file fp the content of the structure pointed to by ptr\_person.

void read\_file(FILE \*fp, person\_t \*ptr\_person);

// This reads from the file fp and puts the bytes into the structure pointed to by ptr\_person.

void seek\_file(FILE \*fp, long offset);

// this decides from which point we start reading from the file or from which point we start writing into the file.

Let us know examine the file **file3.c.**

open\_file gaurantees opening the file in input/output mode. We can read from as well as write to the file. This uses the trick we discussed in ex2.c.

Both read\_file and write\_file have similar structure. The function read\_file calls fread and fills the memory pointed to by the first argument with # of bytes indicated by the second argument. The third argument being 1 indicates that we are reading only one block. The last argument is the file from which we are reading.

The function write\_file does the reverse. It writes into the file, the content of the location to which the first argument points to. The # of bytes written is based on the second argument. The third argument being 1 indicates that we are writing only one block. The last argument is the file to which we are writing.

But there is one question. Where do we write in the file and from where to we read from the file? The internal data structures associated with file opening maintain a field called offset for each opening of the file. Both read and write operations start from this offset. The offset is in bytes.

We can change this offset by using our funciton seek\_file. This funcion in turn calls the function fseek.

Check the man pages.

int fseek(FILE \*stream, long offset, int whence);

We can measure the offset either from the beginning or from the current position or from the end.

The last argument whence indicates it – the values are SEEK\_SET, SEEK\_CUR and SEEK\_END respectively

Observe the second argument. If I want to write the 5th record, it would be in position 4 counting from 0. The number of bytes to skip will be (5 – 1) \* sizeof each record.

FILE\* open\_file(const char\* fname)

{

FILE\* fp;

fp = fopen(fname, "r+");

if(fp == NULL)

{

printf("opening file failed for r+\n");

fp = fopen(fname, "w+");

if(fp == NULL)

{

printf("fatal: cannot open file\n");

exit(1);

}

}

return fp;

}

void close\_file(FILE \*fp)

{

fclose(fp);

}

void write\_file(FILE \*fp, const person\_t \*ptr\_person)

{

fwrite(ptr\_person, size, 1, fp);

}

void read\_file(FILE \*fp, person\_t \*ptr\_person)

{

fread(ptr\_person, size, 1, fp);

}

void seek\_file(FILE \*fp, long offset)

{

fseek(fp, (offset - 1) \* size, SEEK\_SET);

}

Let us examine the client code to create the data structure on the harddisk.

Check the file **ex3.c**.

The file is name is hardcoded. It can as well be a c string – an array of char - to which the filename can be read.

The program supports a menu driven program using the variable opt. Observe we read a value into opt before the loop as well as at the end of the loop.

Let us examine the two options.

1. insert:

The client inputs the name, the age and the position where to write in the file.

We seek to that position based on the variable pos and write to the file.

This sort of organization is called relative file organization.

Let us say that we want to create a file about each of you and want that information to be at a particular position based on your slno in the course. So, we shall have a fixed slot for each of you and we can fill in any order or retrive in any order.

Also observe that this implementation overwrites the record if the same position is given twice.

2. disp:

The client inputs the position. We seek to that position and fetch whatever exists there. If there is nothing written there, we will get junk.

int main()

{

int opt; int pos;

person\_t x;

FILE \*fp = open\_file("three.dat");

printf("0: exit; 1: insert/overwrite; 2: disp : ");

scanf("%d", &opt);

while(opt)

{

switch(opt)

{

case 1 : printf("enter name age and pos : ");

scanf("%s %d %d", x.name, &x.age, &pos);

seek\_file(fp, pos);

write\_file(fp, &x);

break;

case 2 : printf("enter pos : ");

scanf("%d", &pos);

seek\_file(fp, pos);

read\_file(fp, &x);

printf("name: %s age: %d pos: %d\n",

x.name, x.age, pos);

break;

}

printf("0: exit; 1: insert/overwrite; 2: disp : ");

scanf("%d", &opt);

}

close\_file(fp);

}

Take your time. Experiment. Become comfortable before you try the next example.

Can we check before overwriting?

Can we read only if the record has been written?

Can we read all the written records?

We will try to answer a few of these in our next example.

This may require that we keep track of which records are being written.

To do that, we may create a file which has two parts.

The first part is the header which in out example shall be an array of int.

The second part shall be the a sequence of records.

All the elements ofthis array are initialized to 0 indicating that there are no records. Each time we write a record, we also update the position in this array. So, while reading we checck if the element at pos in the array is not 0.

Check out dir4.

The file **person4.h** has a new structure called heade which has an array to indicate which records are filled.

We have also introduced a function to display the structure person.

#define MAX 10

struct header

{

int pos[MAX + 1];

};

typedef struct header header\_t;

struct person

{

char name[20];

int age;

};

typedef struct person person\_t;

extern size\_t size;

void disp\_person(const person\_t\*);

This is the straight forward implementation of the function disp\_person from the file **person4.c.**

void disp\_person(const person\_t\* ptr\_person)

{

printf("name: %s age: %d\n", ptr\_person->name, ptr\_person->age);

}

Let us check the changes in the interface file file4.h.

There have been a few changes compared to file3.h.

We have a new method disp\_all to display all the existing person in the file.

Both read\_file and write\_file have the responsibility of finding the position – they should call seek\_file before reading and writing respectively.

FILE\* open\_file(const char\* fname);

void close\_file(FILE \*fp);

void write\_file(FILE \*fp, const person\_t \*ptr\_person, int pos);

void read\_file(FILE \*fp, person\_t \*ptr\_person, int pos);

void seek\_file(FILE \*fp, long offset);

void disp\_all(FILE \*fp);

Let us examine the implementation file file4.c.

This function init\_header is new. It creates the structure which has an array as the field. All the elements are initaliazed to 0. This whole strucutre is written to the file in the beginning.

This function is invoked from the function open\_file only when a new file is being created.

void init\_header(FILE \*fp)

{

header\_t h;

for(int i = 1; i <= MAX; ++i)

{

h.pos[i] = 0;

}

fwrite(&h, sizeof(header\_t), 1, fp);

}

The call to init\_header is the only change in open\_file. Observe that this is called only once – when the file is new.

FILE\* open\_file(const char\* fname)

{

FILE\* fp;

fp = fopen(fname, "r+");

if(fp == NULL)

{

printf("opening file failed for r+\n");

fp = fopen(fname, "w");

if(fp == NULL)

{

printf("fatal: cannot open file\n");

exit(1);

}

**init\_header(fp);**

}

return fp;

}

void close\_file(FILE \*fp)

{

fclose(fp);

}

Both read\_file and write\_file are changed.

In write\_file, there are two writings on to the file.

The first is to seek to the beginning of the file. Read the header. Modify the entty for the index pos in the array. Seek to the beginning. Write to the file the header.

The second is seek to the right position and write the new record(or overwrite the earlier record).

void write\_file(FILE \*fp, const person\_t \*ptr\_person, int pos)

{

header\_t h;

fseek(fp, 0, SEEK\_SET);

fread(&h, sizeof(header\_t), 1, fp);

h.pos[pos] = pos;

fseek(fp, 0, SEEK\_SET);

fwrite(&h, sizeof(header\_t), 1, fp);

fseek(fp, sizeof(header\_t) + (pos - 1) \* size , SEEK\_SET);

fwrite(ptr\_person, size, 1, fp);

}

In read\_file, we first read the header and check whether the entry in the array at index pos is not 0. If no, we give an error message. If yes, we seek to the position and reaad the record.

void read\_file(FILE \*fp, person\_t \*ptr\_person, int pos)

{

header\_t h;

fseek(fp, 0, SEEK\_SET);

fread(&h, sizeof(header\_t), 1, fp);

if(h.pos[pos] != 0)

{

fseek(fp, sizeof(header\_t) + (pos - 1) \* size , SEEK\_SET);

fread(ptr\_person, size, 1, fp);

}

else

{

printf("no record at that position\n");

}

}

This function disp\_all reads the header. Walks through the array. For every non zero entry in the array, fetches the record and calls the function disp\_person to the display the record.

void disp\_all(FILE \*fp)

{

header\_t h;

person\_t x;

fseek(fp, 0, SEEK\_SET);

fread(&h, sizeof(header\_t), 1, fp);

for(int pos = 1; pos <= MAX; ++pos)

{

if(h.pos[pos] != 0)

{

fseek(fp, sizeof(header\_t) + (pos - 1) \* size , SEEK\_SET);

fread(&x, size, 1, fp);

disp\_person(&x);

}

}

}

Trust the above examples give you an idea how you can create data structures on hard disk.

Try modifying our array list implementation to create list on harddisk. You have the header followed by # of nodes.

You may await your 4th assignment or project by Monday.