CIS*2520 Assignment 1

Due: Sun, Sep 29, 11:59 PM

Version 1.04 (changed highlighted in yellow)

Introduction

For this assignment, you will be comparing the performance of linked lists vs arrays. For the first part of the assignment, you will be writing code to develop a memory management system in a binary file on disk, a linked list library, an array library, and code to perform various operations on the two data structures. For the second part, you will write a report detailing the performance differences between the two implementations.

Part A - Coding

Memory management system

For this part, you will create code to manage a binary file. Think of the binary file as a large block of memory, or as an array.

Create a file, "ds_memory.c" and a complementary "ds_memory.h". Implement the following constants, structures, variables, and functions in these files:

```
MAX_BLOCKS      4096
```

This constant indicates the maximum number of blocks of memory that we will track in the file. Define this in the header file.

```
struct ds_counts_struct {
     int reads;
     int writes;
};
```

This data structure should count the number of read and write operations performed (see below).

```
struct ds_blocks_struct {
     long start;
     long length;
     char alloced;
};
```

This is a structure that keeps track of one block in the binary file. Define this in the header file.

```
struct ds_file_struct{
     FILE *fp;
     struct ds_blocks_struct block[MAX_BLOCKS];

};
```
This is a structure that holds a file pointer to a file, open in binary, read/write mode, and an array of blocks. Define this in the header file.

```
struct ds_file_struct ds_file;
struct ds_counts_struct ds_counts;
```
These are two global variables. The first holds the file pointer and the blocks array, the second the read and write counts. Note that you should not use **ANY** other global variables, and should **NEVER** try to load the entire binary file into memory. Access to the binary file's contents should be only via the read and write operations, below. Define this in the c file.

`int ds_create( char *filename, long size )`
This function should create a file with the provided **`filename`**. The file should consist of a header which is represents the **`block`** array. The first entry in the array should have a **`start`** value of 0, a **`length`** value equal to **`size`**, and an **`alloced`** value of 0**.** All other entries in the header should have a **`start`** value of 0, a **`length`** value equal to 0, and an **`alloced`** value of 0. Following the header, the file should consist of an additional **`size`** bytes with all bytes set to zero. The function should close the file upon completion. The function should return the value 0 if the function succeeds, and a value other than zero if anything goes wrong. Define this, and all other functions in the c file.

`int ds_init( char *filename )`
This function should open the given filename in binary read/write mode, save the file pointer in the global variable and load the header into the global variable. It must **not** load any other parts of the file. This function should set **`reads`** and **`writes`** to 0 in the **`counts`** variable. If successful it should return 0, otherwise a value other than 0.

`long ds_malloc( long amount )`
This function will search through the **`block`** array (from lowest index to highest) until it finds a block with **`length`** larger than, or equal to **`amount`**, and **`alloced`** equal to 0. This is the first block that is unused and large enough to hold **`amount`**. When it finds such a block, it will change the **`length`** of the block to **`amount`**, and **`alloced`** to 1. This indicates that the appropriate **`amount`** of memory has now been allocated. It will search (from lowest index to highest) for a second block with **`length`** equal to 0 and set its **`start`** value to the original block's **`start`** value plus **`amount`**, set its **`length`** to the original block's **`length`** value minus **`amount`**, and **`alloced`** to 0. This will represent the left-over memory of the original block. The function should return the **`start`** of the first block found (i.e. the newly allocated block). If no un-**`alloced`** block of sufficient size can be found, the function should return the value of -1. If no second block with **`length`** 0 can be found, that's ok. Do not sort the blocks.

Do not move the blocks. Do not consolidate blocks. (If this were an operating systems course, I might have made you do that.)

`void ds_free( long start )`
This function will search through the **`block`** array (from lowest index to highest) until it finds a block whose **`start`** value matches **`start`**. It will then set the value **`allocated`** value of that block to 0. If no block is found the function should return without doing anything.

`void *ds_read( void *ptr, long start, long bytes )`
This function should read **`bytes`** bytes from the location given by **`start`**, offset by the length of the header in the binary file. This should increment the value of **`reads`** in the **`counts`** variable by 1. If successful, it should return the address **`ptr`**, if unsuccessful it should return **`NULL`**.

`long ds_write( long start, void *ptr, long bytes )`
This function should move the file pointer to the **`start`** location offset by the length of the header in the binary file, and write **`bytes`** bytes to the file from the address **`ptr`**. This should increment the value of **`writes`** in the **`counts`** variable by 1. If successful, the function should return a value of **`start`**. If something goes wrong, it should return a value of -1.

`int ds_finish()`
This function should write the **`block`** array into the header at the beginning of the file and close the file. It should print out the values of the counts data structure exactly as follows:

```
reads: 27
writes: 22
```

Note there is a single space after the colon and a single newline character after the number. I.e. the first line is 10 characters in length, and the second is 11 characters in length. If successful it should return 1, otherwise 0. Note that this is the only printing that should be done in any parts of the memory management system (remember to comment out your debugging print statements).

Array management system

For this part, you will create code to manage an integer array that lives in the binary file that the previous part manages. Note: you should never load the entire array into memory, instead you should access individual elements, one at a time, reading and writing as needed and described below.

Create a file, “ds_array.c” and a complementary “ds_array.h”. Implement the following constants, structures, variables, and functions in these files:

`MAX_ELEMENTS    256`
This constant represents the maximum number of elements that can be included in the array.

`long elements;`
This global variable will hold the number of elements currently in the array.

`int ds_create_array()`
~~This function should create an array on disk by calling **`ds_create`**, above, to hold a single **`long`** plus **`MAX_ELEMENTS`** of **`int`** in a file called "array.bin".~~ It should call **`ds_init`**, above, to open the file, "array.bin". This function should **`ds_malloc`** memory for the single **`long`** at the beginning of the binary file. It should set the value of that **`long`** to 0 to indicate that there are no elements currently in the array. It should then call **`ds_malloc`** (once) to allocate data for the entire array. Finally, it should call ds_finish, above, to close the file and print the number of reads and writes. If successful, it should return 0, otherwise a value other than 0.

`int ds_init_array()`
This function should access an array on disk by calling **`ds_init`**, above, to open the file "array.bin". It should read the first block of memory, which holds a **`long`**, and store that value in the global variable, **`elements`**. If successful, it should return 0, otherwise a value other than 0.

`int ds_replace( int value, long index )`
This function should replace the contents located at the given **`index`** in the array with **`value`**, by calling **`ds_write`** with the appropriate parameters. It should return 0 if successful and a value other than 0 if unsuccessful (e.g. invalid index).

`int ds_insert( int value, long index )`
This function should store the contents located at the given **`index`** in the file in a temporary variable, while inserting the value **`value`** at that location, by calling **`ds_read`** and **`ds_write`** with the appropriate parameters. Then it should insert the value that was stored in the temporary variable at the next index in the array, and continue moving all subsequent elements along to the end of the array, incrementing the value of **`elements`** by 1. This function should be able to insert a new element at the end of the array (**`index==elements`**). It should return 0 if successful and a value other than 0 if unsuccessful (e.g. invalid index, exceeding **`MAX_ELEMENTS`**).

`int ds_delete( long index )`
This function should remove the item at **`index`**, moving all other items forwards by one index and reducing **`elements`** by 1. It should return 0 if successful and a value other than 0 if unsuccessful (e.g. invalid index).

`int ds_swap( long index1, long index2 )`
This function should swap the elements stored at **`index1`** and **`index2`**. It should return 0 if successful and a value other than 0 if unsuccessful (e.g. invalid index).

`long ds_find( int target )`
This function should search the array for the first element whose value is equal to **`target`** and return the index of the element or -1 if no element matches **`target`**.

`int ds_read_elements( char *filename )`
This function should **`fscanf`** elements of type int from a text file with name **`filename`**. The text file will have one integer per line. Each element should be inserted into the array using the **`ds_insert`** function. The order of elements in the array should match the order of elements in the file. It should return 0 if successful and a value other than 0 if unsuccessful (e.g. file errors, exceeding **`MAX_ELEMENTS`**).

`int ds_finish_array()`
This function should write the value of **`elements`** to the beginning of the binary file, and call **`ds_finish()`**. It should return 0 if successful and a value other than 0 if unsuccessful.

Linked list management system

For this part, you will create code to manage a linked list holding integers that lives in the binary file the first part manages.

Create a file, "ds_list.c" and a complementary "ds_list.h". Implement the following constants, structures, variables, and functions in these files:

```
struct ds_list_item_struct {
     int item;
     long next;
};
```
This holds a single integer from the list and the binary file location of the next item.

`void ds_create_list()`
This function should call **`ds_init`** with the filename "list.bin", and **`ds_malloc`** memory for a single **`long`** in a binary file called "list.bin". It should set the value of that **`long`** to -1 to indicate that there are no more items in the list. It should call ds_finish. It should return 0 if successful and a value other than 0 if unsuccessful (e.g. bad filenames).

`int ds_init_list()`
This function should access a list on disk by calling **`ds_init`**, above, to open the file "list.bin".

`int ds_replace( int value, long index )`
This function should traverse the list beginning with the element indicated by the initial integer in the binary file until it has gotten to the element at index **`index`** or reached the end of the list (**`next`**==-1). If it reaches **`index,`** it should replace the contents located at the given **`index`** in the list with **`value`**, by calling **`ds_write`** with the appropriate parameters. It

should return 0 if successful and a value other than 0 if unsuccessful (e.g. invalid index, reached end of list).

`int ds_insert( int value, long index )`
This function should traverse the list beginning with the element indicated by the initial integer in the binary file until it has gotten to the element just before index **`index`**. It should change the **`next`** value of that element to point to a new element created by calling **`ds_malloc`**. The new element should be populated with the value **`value`** and its **`next`** attribute should be set to connect to the remainder of the list. This function should work properly if **`index`** is 0 (i.e. it should modify the first integer in the binary file). It should return 0 if successful and a value other than 0 if unsuccessful (e.g. invalid index, out of memory).

`int ds_delete( long index )`
This function should remove the item at **`index`**, keeping the rest of the list intact. This function should work properly if **`index`** is 0 (i.e. it should modify the first integer in the binary file), or the last item in the list. It should return 0 if successful and a value other than 0 if unsuccessful (e.g. invalid index).

`int ds_swap( long index1, long index2 )`
This function should swap the elements stored at **`index1`** and **`index2`**. It should return 0 if successful and a value other than 0 if unsuccessful (e.g. invalid index).

`long ds_find( int target )`
This function should search the array for the first element whose value is equal to **`target`** and return the index of the element or -1 if no element matches **`target`**.

`int ds_read_elements( char *filename )`
This function should **`fscanf`** elements of type int from a text file with name **`filename`**. The text file will have one integer per line. Each element should be inserted into the array using the **`ds_insert`** function. The order of elements in the array should match the order of elements in the file (the first item in the file should be at the head of the list). It should return 0 if successful and a value other than 0 if unsuccessful (e.g. file errors, exceeding total memory).

`int ds_finish_list()`
This function should call **`ds_finish()`**. It should return 0 if successful and and a value other than 0 if unsuccessful.

Makefile

Write a makefile that compiles your .c files into .o files. You may include other build instructions in your makefile for test programs that you will not submit via git.

Part B – Analysis

Write some programs to use and test all your functions. You should create a program that fills a text file with a number of random integers in the range from 0 to 100. This program should be able to create files with lots of numbers of very few numbers.

1. Now, create a file with 10 numbers and use your linked list implementation and chose a random index and **`ds_replace`** the value there with the value 0. Repeat this 30 times, recording the reads and writes. Now do it again with a file that has 20 numbers, and 30 numbers, 40 numbers, … all the way to a file with 100 numbers. Since this will involve 300 experiments in total, I recommend you writing some code to do this. You can write the code in c if you like, or a different language (you will not hand it in). Draw a graph with the number of numbers in the file (lines in the file) on the x-axis, and the number of reads or writes on the y-axis. Use different colours for reads and writes. Label your axes. Provide a key. Give your figure a title. You can plot minimum, maximum and average values for each count (6 coloured lines total: rd_max, rd_avg, rd_min, wr_max, wr_avg, wr_min), or for a small bonus, use a box-whisker plot (google it).
2. Repeat step 1 with the array implementation.
3. Repeat step 1 with linked list implementation and the **`ds_insert`** operation (instead of **`ds_replace`**).
4. Repeat with array and the **`ds_insert`** operation.
5. …

Create all of the 8 graphs in the following table.

Graph #	**Operation**	**Implementation**
1	ds_replace	List
2	ds_replace	Array
3	ds_insert	List
4	ds_insert	Array
5	ds_delete	List
6	ds_delete	Array
7	ds_swap	List
8	ds_swap	Array

Discussion: Come up with some practical examples where you think (based on your graphed results and your understanding of the two data structures) a linked list might be a more efficient data structure to use than an array, based on typical use cases. Do the same for some examples where the array would be superior.

Put your graphs and your answer to the **Discussion** above in a PDF file (not text, not Word, not Pages).

Submission

You will submit all of your assignment's files by depositing them to the School's git repository in the A1 submission directory. Submit exactly the following files:

makefile, ds_memory.c, ds_memory.h, ds_array.c, ds_array.h, ds_list.c, ds_list.h, analysis.pdf

Put your full name, student ID number, and uoguelph e-mail at the top of each file.

Use consistent indenting, formatting, commenting, naming and other good programming practises to make your code readable.