Jacob Khalili
Gary Kim
Aliza Meller
ECE-251
Professor Billoo
4/7/21

Computer Architecture ECE-251: Project 2

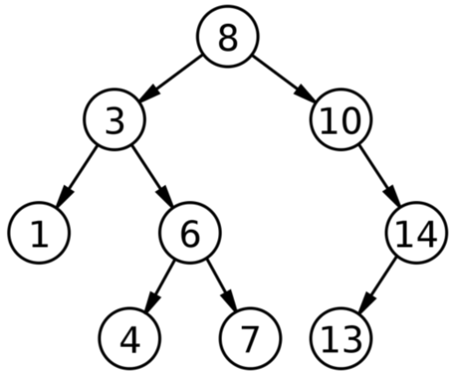
Requirements: Read a file the contains a list of numbers. One number per line—limited to 32 bits and each line is terminated by '\n'. Create another file that has the numbers in ascending order.

Overall Architecture of Program

After declaring all the initial variables in the .data section, a loop is used to read and sort each number in the input file using a binary tree sort algorithm. The first number in the file is the root node and the following numbers are sorted to left or right depending on whether they are greater than, less than or equal to the preceding numbers in the tree. After re-ordering the numbers, fopen is used with the "w" parameter to write a file containing the sorted list. Along the way, progress messages are displayed to indicate which steps in the process of opening, reading and writing the files were completed.

Overall Algorithm Design:

A binary search tree is used to search to store and search the integers provided. Each node on the tree stores the following information in this format:



A binary search tree is used in order to allow for simultaneous reading, storing and sorting of the data. This also allows for dynamic allocation of memory in the heap. This help avoid the issue of running out of memory as we read in more data from the input file. We use malloc to do this dynamic memory allocation.

Since assembly does not have the built-in idea of a struct, we created our own by having an agreed upon offset for each property of a treenode. The agreed upon offsets were the following (it did turn out on further research that we may have allocated more memory than needed for the integer):

Jacob Khalili
Gary Kim
Aliza Meller
ECE-251
Professor Billoo
4/7/21
We declare a tree structure according to the following convention:

Bytes:	Data:
0-7	Integer (Max 32 Bits)
8-15	Pointer to Left Node
16-23	Pointer to Right Node

The nodes are arranged in order: for each node, all elements in its left subtree are less-than the node (<), and all the elements in its right subtree are greater-than or equal to the node (≥). Recursively, each of the subtrees must obey these constraints as well.

Design Decisions:

Originally, we intended to create an array of values then sort it with bubble sort. This presented a couple difficulties. For one, writing the code to be able to know ahead of time how much memory would have to be created for an array was complex. Writing a bubble sort would also have been complex. We went through several ideas for how to both store and sort the values including using a linked list. We eventually settled on using a binary search tree as that would allow us to both sort and store the data in one go without having to write separate code for each process and it meant we didn't need to know the size of the input beforehand as we could allocate memory to store the data from the heap at runtime.

Besides for using fscanf, we chose to use fclose to give the user the ability to overwrite the file if they choose. This could be done by calling the naming the output file the same as the input file.

Limitations:

There are a couple limitations with the program. One of them is that the maximum and minimum values that it can handle are the max and minimum values that can be stored in 4 bytes (INT_MAX and INT_MIN). Another is that the program cannot be called on a file that has no numbers. Otherwise, the output file will simply contain a "1".

.data Section

Name	**Use**
inputFile	Store the name/location of the input file
outputFile	Store the name/location of the output file

Jacob Khalili
Gary Kim
Aliza Meller
ECE-251
Professor Billoo
4/7/21

programName	Stores the name of the program for use in the help message
helpMessage	Stores the help message
readMode	“r” – used in reading the file
writeMode	“w” – used in writing the file
templateString	“%d\n” – used in reading the file
integer	where the value from scanf is stored
treeRoot	Location in RAM of the treeRoot
Newline	“\n”
filepointer	Used in reading/writing to the file
Remainder	Are used for debugging

Main:

The contents of r0 contain the number of arguments used when executing the program. First, r0 is compared to #3. This allows for the program to check is the expected number of arguments were included when the program is executed. Before it branches to another section, the address of programName is loaded into r5 and the value pointed to by the stack pointer incremented by 8 bytes is loaded into r3. This value represents argv[0], the name of the program. The contents of r3 are then loaded into r2, which is then stored in the address pointed to by r5, namely, the address of the programName. As a result, the address of programName contains argv[0]. So, if there are less than 3 arguments, meaning that the user did not include either the input file or output file, the program branches to the printHelpMessage section which uses the value of argv[0] and will be explained in more detail below.

The same process of storing argv[1] and argv[2] into the proper addresses in memory were repeated for the following two arguments, the input file name and the output file name. The file is then opened by loading the address of the inputFile into r0, the value of the input file name into r0, loading the address of readMode into r1 and utilizing the fopen function. With the use of readMode, the fopen function reads a file and returns a pointer to a file object associated with the named file in r0. Therefore, when r0 is compared with #0, if it is equal, that

Jacob Khalili
Gary Kim
Aliza Meller
ECE-251
Professor Billoo
4/7/21

means that the program did not successfully read the file and it branches to the cannotOpenFile section which will be explained in more detail below.

If the program did read the file successfully, it continues linearly to the next instruction in main in which the address of the filePointer is loaded into r5 and then the contents of r0 which contains the file pointer is stored at that address.

To maintain the registers used thus far and avoid possible overwrites, the registers r0 through r12 and the link register are pushed onto the stack. The address of progress1 (contains the message that the file has been read) is loaded into r0 and the printf function is called to print that message. The saved registers are then popped off the stack.

The first number is then read from the file. The address of templateString is loaded into r1 and the address of integer is loaded into r2. These two registers are used as the arguments for the fscanf function which is called to read the first number in the file.

Lastly, the treeRoot is created. #24 is moved into r0 and then malloc is called to allocate 24 bytes of memory and the pointer to that allocated space is returned in r0. The address of treeRoot is loaded into r8 and then r0, which contains the pointer to treeRoot, is stored in the address of treeRoot. The address of integer is loaded into r6 and the integer that was read is loaded into r6. The integer is then stored in the address pointed to by r0****. The struct format for treeNode establishes that bytes 0-7 are the integer, bytes 8-15 are the left node and bytes 16-23 are the right node. As a result, #8 is added to r0 and placed in r0 to point to the pointer to the node on the left. #0 is moved into r7 and then the contents of r7 are stored in the address pointed to by r0 to guarantee that r0 is zeroed out. Similarly, for the right node, #8 is added to r0 and stored in r0 and then the contents of r7 are stored in the address pointed to by r0.

read1:

The next integer in the file is read similarly to the way the first integer is read, which acts as the root node in the tree. The file pointer is loaded into r0, the address of templateString is loaded into r1 and the address of integer is loaded into r2. Then, in order to read the next number from the file, we leveraged the fscanf function.

The program first checks if the integer read is an actual value. This is done by comparing r0, which contains the number of bytes read from fscanf to #1. If there was nothing read, fscanf would contain #0, if one number was read, fscanf would contain #1. As a result, if r0 is not equal to #1, meaning that fscanf did not read an actual value, the program branches to end1 which will be discussed in more detail below.

Jacob Khalili
Gary Kim
Aliza Meller
ECE-251
Professor Billoo
4/7/21
If r0 contains #1, the program continues linearly to add the value to the tree. This is accomplished by loading the address of treeRoot into r0, which contains the pointer to the area in memory allocated for the tree by malloc, and loading the integer into r1, which contains the integer that was read by scanf. The program then branches to store, which will be discussed in more detail below.

end1:

The purpose of the end1 section is to write the sorted numbers to a file. The file pointer to the input file is loaded into r0 and the file is closed using the libc function fclose. Registers r0-r12 and lr are pushed onto the stack to avoid overwriting them in the next instruction when the progress2 message is printed into the stdout using the libc function printf. The registers are then popped off the stack and the program proceeds to the next step of opening a file to write to.

In order to open a file to write to, the libc function fopen is called after outputFile, which contains the name of the outputFile (argv[2]), is loaded into r0 and the address of writeMode is loaded into r1. To check that the fopen was successful is compared to #0. If fopen is successful the contents of r0 should contain a pointer to the file if successful, so if r0 contains #0, fopen failed. As a result, the cmp instruction is used followed by the beq instruction which branches to the cannotWriteFile if r0 is equal to #0. This section will be explained in more detail below. If r0 is not equal #0, meaning fopen is successful, the program continues linearly to the next instruction.

Before calling printNode, the registers r0-r12 and lr are pushed onto the stack. r0, which at this point contains the file pointer to the output file, is moved into r1 and the address of treeRoot which contains the pointer to the space allocated in memory is loaded into r0. Afterwards, the program branches to the printNode section, which determines whether the next integer should be placed leftwards or rightwards of the first number read. After printNode is finished performing its respective tasks, the program branches back to the place it left off from in end1 and then registers are popped back from the stack.

To avoid overwriting the registers, they are pushed back onto the stack, after which the address of newLine is loaded into r1 and fprintf is called to insert a newline character after each number. The registers are then popped back from the stack and fclose is called to close the file that is being written to.

The registers are pushed back onto the stack and then the last message indicating that the file was successfully written to (progress3) is printed to stdout. This is accomplished by loading the address of progress3 into r0 and calling the libc function, printf. The registers are then popped back from the stack, and the program exits by moving the return code (#0) into r0, moving #1 into r7 (which is the swi system call to exit out of the program) followed by ‘swi 0’.

store:

Store a function that contains the algorithm for inserting a new node with a given integer into the binary search tree. It is designed to be run recursively as trees are easiest to implement in a recursive manner. The function should be called with r0 being the pointer to the position in memory with the treenode and r1 being the integer that we wish to add to the binary search tree.

The function first loads the content at r0 into r5 to use later. r5 is then loaded back into r0. At this point, r0 is the integer at the node that the function was called with. r0 and r1 are then compared, keeping in mind that at this point, r1 is the integer we wish to add to the BST. r5 is then moved back into r0 so that r0 is the location of the treenode again. At this point, 8 is added r0 if the compare was greater than or equal. If it was less than, 16 is added to r0. This moves the value in r0 from being the location of the start of the treenode in memory to being the location of the pointer to either the left or right treenode depending on if the value to be added is larger than or smaller than the value at the current treenode.

After this, it checks if the leaf is null. If the leaf is null, the current r0 and lr will be pushed onto the stack then the program will jump to “storeContinue”. If not, registers r0-r12 and lr are pushed onto the stack then store function is run again (this is the recursive call). At this point, the r0 is now a pointer to a point in memory that contains the location of the next treenode. After the store call returns, the pushed values are popped again.

After this, “bx lr” is called to return the function.

storeContinue:

storeContinue is the function that handles creating new treenodes. To begin, it moves r0 to r8 and r1 to r6 to keep those values. After this, #24 is put into r0 then malloc is called. Malloc will then allocate 24 bytes of memory from the heap and return the address of it. The address of it will be saved into the location referenced by the value in r8. This sets the leaf property of the parent node to be the treenode that we are now creating. The memory address is then also moved to r5.

At this point, we start putting the data into the new treenode. First, we save the integer stored in r6 (which is the integer we want to add to the BST) into the memory location stored in r0. After this, we set both leaves to be null (necessary as it is not guaranteed that malloc will return memory that has been zeroed out). To do this, the value at r0 is moved into r6 for use later. #8 is added to r0 to get to the point in memory that is the left leaf on the treenode. r7 is set to #0. The value of r7 is stored into the location referenced in r0. After this, r0 is set to r6 + #16. The value of r7 is saved into the location refereced by r0. After this, r0 and lr are popped from the stack then the function returns with “bx lr”.

printNode:

This algorithm works recursively to print the entire the tree structure in order. It prints the left node (if it exists), itself, and the right node (if it exists)

The code starts by checking if there is a left side, if there is it calls printNode on the left side. After than it prints its own value. If there is a node to the right, it calls printNode on that. Due to the structure of the tree this will print an ordered list.

printHelpMessage:

As mentioned above, the program only branches to this section when the user does not execute the program with three arguments. The help message provides directions on how to properly execute the program, the usage of the program and an example of correct execution with 3 proper arguments including the name of the program.

The address of programName and its contents are loaded into r1, which contains the value of argv[0] (i.e. the name of the program that is being executed which can change based on user preferences). This value is then moved into both r2 and r3.The address of helpMessage, which is declared in the .data section, is loaded into r0 and then printf is called to print the help message with the specific program name to the stdout. The program then exits using a system call with the exit code 0. This is accomplished by moving #0 in r0, moving #1 into r7 and using the ‘swi 0’ instruction.

cannotOpenFile:

As mentioned above, if the program does not successfully open the file, after fopen is called with readMode, no file pointer would be returned and the value in r0 would be 0. As a result, the program would branch to cannotOpenFile. The cannotOpenFile section prints a message to stdout that the file could not be opened.

This is accomplished by loading the address of cannotOpenFileMessage, which is declared in the .data section, into r0, loading the input file name into r1 and calling the printf function to print the message which includes the specific file name. The program then exits using a system call with the exit code 0. This is accomplished by moving #0 in r0, moving #1 into r7 and using the ‘swi 0’ instruction.

Jacob Khalili
Gary Kim
Aliza Meller
ECE-251
Professor Billoo
4/7/21
cannotWriteFile:

Similar to cannotOpenFile, the cannotWriteFile section prints a message that the output file provided cannot be written to. The program only branches to this section from the end1 section if the contents of r0 is equal to #0 after calling the fopen function with writeMode. This means that there was no pointer to the output file returned.

This is accomplished by loading the address of cannotWriteFileMessage, which is declared in the .data section, into r0, loading the output file name into r1 and calling the printf function to print the message which includes the specific file name. The program then exits using a system call with the exit code 0. This is accomplished by moving #0 in r0, moving #1 into r7 and using the ‘swi 0’ instruction.