CSI 370 Computer Architecture
Towers of Hanoi in Assembly

Reuben Kernan   Alex Taxiera

April 29, 2018

1 Introduction

1.1 Summary

The flagship of this project is the implementation of the Towers of Hanoi algorithm using Assembly. Supplementary challenges include building an implementation of a stack, as well as building an implementation of a queue. These challenges were originally assigned as bonus problems during the 'Data Structures & Algorithms' course, where implementation was done in C++. This is not a direct port of the original implementation, rather, implementation is performed from the ground up. The challenge is completed using MASM32.

1.2 Goals

- Implement a stack data structure
- Implement a queue data structure
- Solve the Towers of Hanoi problem

1.3 Towers of Hanoi Concepts [1]

Rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3. No disk may be placed on top of a smaller disk.

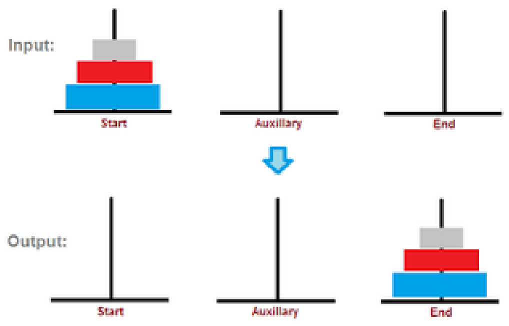

Figure 1: Towers of Hanoi initial and end states.

1.4 Challenges

1. The complexity of Towers of Hanoi scales exponentially with the number of disks to be transferred. The minimum moves required is: '$2^n - 1$' where 'n' is the number of disks. If the solver could perform 1,000 moves per second, and there were 100 disks, then it would take approximately 4.0×10^{19} years to solve. Clearly, performance is of utmost importance when attempting to solve Towers of Hanoi with an increasing number of disks. For this reason, registers are used often as possible in the implementation due to their speed of execution compared to variables which will be physically further in memory storage.

2. Preserving values saved within registers presents a significant problem, particularly when using recursion, or procedures that invoke other procedures. For example, changing the ECX register inside the _stackPush function could affect the initialize_loop in _main. Due to challenge 1, no effort is made to save off registers before invoking procedures in an attempt to increase efficiency. Therefore, all code is very mindful of which registers are used or can be overwritten.

3. Output is foregone entirely within this program, making it difficult to assess functionality. The program is tested manually on a small scale, where _maxHeight is initialized to 10, and _diskNum to 3. Testing is described in section 3.

4. OURSTACK and OURQUEUE are not intrinsically linked to their respective procedures. This means that their contents are bared to the

entire program, allowing it to access elements in a way not intended. Two implementations of a queue are demonstrated, one operating directly on OURSTACK to mimic the behavior of a queue, and the other using its own separate procedures and STRUCT definition. More on this topic in section 2.2.

2 Implementation

2.1 Stack

```
_stackPush PROTO C, Stack:DWORD, value:BYTE
_stackPop  PROTO C, Stack:DWORD
.DATA
OURSTACK STRUCT
    height    BYTE 0
    levels    BYTE _maxHeight DUP(0)
OURSTACK ENDS
```

OURSTACK is declared within the .DATA segment. This enables any number of instances to be created in the code, where all values are initialized to 0. For the niche purpose of solving the Towers of Hanoi, each data point used occupies only a single byte. The maximum value for the height of a stack is 255 elements, allowing representation of more than enough disks in Towers of Hanoi to occupy Multivac [1]. The _maxHeight global variable is used as a single access point to infinitely scale the number of elements that can be stored in OURSTACK, though technically height could overflow if more than 255 elements are pushed.

There are two companion procedures to OURSTACK. _stackPush is used with INVOKE to pass in a pointer to an instance of OURSTACK and a value to be pushed onto it. It attempts to push, but returns -1 in the AL register if the stack is full. _stackPop is used with INVOKE to pass in only a pointer, and returns -1 if the stack is empty.

This stack provides the bones for representing the three towers used in Towers of Hanoi: Source, Auxiliary, and Destination. OURSTACK was designed specifically as a companion to the Towers of Hanoi problem, thus

[1]From a classic science fiction story by Isaac Asimov. Can be found here: http://www.multivax.com/last_question.html.

informing the design process. Maximum number of disks is limited to 255 out of practicality, any more would take too much computing power. Disks are initialized sequentially starting from _maxHeight (biggest) to 1 (smallest), therefore levels only holds byte-size elements since the largest a disk can be is 255.

2.2 Queue

2.2.1 Fake Queue

```
_stackDequeue PROTO C, Stack:DWORD
```

The _stackDequeue procedure is implemented to give a queue-like feature to the OURSTACK struct. This function removes the element that is at the bottom of the stack, returns it, and then shifts every element down so that the stackl can fill in the missing hole. This demonstrates the unprotected nature of the STRUCT implementation.

2.2.2 Real Queue

```
_enqueue PROTO C, Queue:DWORD, value:BYTE
_dequeue PROTO C, Queue:DWORD
.DATA
OURQUEUE STRUCT
    Input  OURSTACK <>
    Output OURSTACK <>
OURQUEUE ENDS
```

OURQUEUE is declared in the .DATA segment. Like OURSTACK, any number of instances can be created, and all values are initialized to 0. This queue implementation uses two stacks to represent the queue. As it is built on the OURSTACK implementation, it is subject to the same limitations. The Input stack initially holds any element that is enqueued, or pushed onto it. When a dequeue is requested, and if the Output stack is empty, then a loop pops all elements from Input and pushes them onto Output, reversing their order in the stack, after which one element is popped from Output. For consistency with OURSTACK, the number of elements in the queue is artificially limited to _maxHeight, even though it could contain twice as many elements as OURSTACK.

There are two companion procedures to OURQUEUE. _enqueue is used with INVOKE to pass in a pointer to an instance of OURQUEUE and a value to be pushed onto it. It manipulates the two stacks as described above, and through he AL register it returns either the first value that was placed into the queue, or -1 to indicate the queue is empty. It is _enqueue's responsibility to enforce the artificial size limit. _dequeue is used with INVOKE to pass in only a pointer, and returns -1 if the queue is empty. _dequeue is more computationally heavy than _enqueue, as it is responsible for moving every element from Input to Output.

OURQUEUE is not relevant in the implementation of Towers of Hanoi, it is simply here as a bonus solution.

2.3 Towers of Hanoi

```
_towers PROTO C, disks:BYTE, Source:DWORD,
                 Destination:DWORD, Auxiliary:DWORD
```

_towers is the implementation of the Towers of Hanoi algorithm. Despite the computational complexity of the problem, the solution in contrast is quite simple and elegant. This procedure is merely 25 lines long, including comments and whitespace. _towers is used with INVOKE to pass in the number of disks, and three pointers to instances of OURSTACK, being Source, Auxiliary, and Destination. Note that the order in which towers are passed into _towers changes throughout the recursive calls [lines 461-463], therefore the original towers can act in different roles. If disks is 1, then it skips to swap, making no more recursive calls, else three calls to _towers are made, passing towers in different orders each time.

3 Testing

3.1 General

_main is the test stub for this project. It is here where simulations run to test the procedures for the OURSTACK and OURQUEUE structs as well as the Towers of Hanoi problem. Four tests are run, one for general OURSTACK operations, one for general OURQUEUE operations, one for the _stackDequeue procedure, and one to solve the Towers of Hanoi problem. If a test encounters an invalid value, such as the disks on the destination tower in the Towers of Hanoi test being in the wrong order, the process will exit with a non-zero code indicating which test failed. This however does not allow to see if multiple tests failed, but works well enough.

3.2 Towers of Hanoi Speed

In addition to testing all the functional aspects, tests are run to see how a computer[2] could complete the Towers of Hanoi problem with varying levels of complexity. To time the process Visual Studio's built in diagnostics tool is used which can give the amount of time used between breakpoints while debugging.

The first trial is run with only twenty disks, this is small enough to be done quickly and get a feel for how fast it can be executed. This trial takes only thirty-two milliseconds, and later trials a bumped up to the thirties for further trials. Thirty disks takes about thirteen seconds to complete, with thirty-five taking nearly seven minutes to solve.

Disks	Time
20	32ms
30	12,762ms
31	25,665ms
32	53,454ms
33	107,961ms
34	199,475ms
35	418,707ms
100	1.3×10^{15} years

Table 1: Speed tests (100 disk calculation based on a single trial.)

[2]Tests were run on an Intel i7-4770k at 4.3GHz

References

[1] Patel, J. (n.d.). Tower Of Hanoi. Retrieved from http://javabypatel.blogspot.in/2015/12/tower-of-hanoi.html