Seth Gellman (sgellm01), Jack Adkins (jadkin05)
11/9/24
CS40

HW06 UM Design Document

Architecture

Module 1: command line file handling (main)

- Description:
  - Create a file to handle our main function and ensure that the file given is valid
- Input:
  - A file containing 32 int instructions
- Relations:
  - Calls a function from the IO interface to read in all of the information from the file provided
- Output/Return Value:
  - EXIT_SUCCESS or EXIT_FAILURE
- Notes:
  - Exits program if there is not one command line argument or if the file on the command line cannot be opened using fopen
- Hanson/External Structures:
  - Uses the stat struct, located in sys/stat.h, to collect the total size of the file being read in

Module 2: IO

- Description:
  - Read in all information from the user, either from stdin or the command line file
- Input:

    - At first, the file from the command line will be passed in to read
    - Later on, it will read in a char from stdin when the input command is given, or print out a char if the output command is given
- Relations:
    - Will be called from main (module 1) and can be called from our instructions interface (module 3) if the input or output command is triggered
    - Uses the memory module to initialize the sequence of all segments (represented as UArrays) to store instructions in segment 0
    - Calls the command loop to begin handling instructions
- Output/Return Value:
    - read_in returns the sequence of segments with all of the file information in segment 0
    - If called, output the char
- Notes:
    - May exit the program if the file ends prematurely
- Hanson/external Structures:
    - Uses the stat struct described in module 1

Module 3: Execution of the 14 operators/instructions

- Description:
    - A file that contains the individual functions for executing the operator code on the registers
- Input:
    - Any registers needed as input for the individual operators
    - The struct containing all relevant register values and segments (explained below module 5)
- Relations:
    - Input or output will call a corresponding function in IO
    - Command loop will call any necessary instruction in the interface
- Output/Return Value
    - Nothing, the output function will export its purpose to IO

- Notes:
    - None
- Hanson/external Structures:
    - None

Module 4: Memory Management

- Description:
    - Handles all memory related functions, most importantly mapping and unmapping segments
    - Also initializes the hanson Sequence of uarrays, which is intended to represent all of the segments in a list
- Input:
    - The length of a segment in bytes
- Relations:
    - Is utilized by both the IO and instructions interfaces for the initialization of the segment Sequence and for mapping or unmapping segments
- Output/Return Value:
    - Initialization will output the sequence of segments
    - No output when mapping/unmapping segments
- Hanson/External Structures:
    - We are handling all segments of our UM as a Hanson sequence with each element representing a single segment
    - Each individual segment will be represented as a UArray since we will know the size needed before initializing it.
    - Hanson Sequence of Hanson UArrays
- Notes:
    - Will use a table and integer to reuse freed blocks (later explanation in implementation)

Module 5: Command Loop

- Description:
    - Will go through each command and call each subsequent instruction
    - Uses a struct with an int determining each time which segment the instruction will be read from
        - We will detail the contents of the segmentData struct in the implementation plan
- Input:
    - Struct containing the the sequence of all segments along with an identifier for what segment we want to read an instruction from
- Relations:
    - Will call the functions in the executions of instructions/operators module
    - The command loop is called in the IO module
- Output/Return Value:
    - None
- Hanson/External Structures:
    - Reads in from the instructions in a uarray representing segment 0 of the sequence
    - Accesses the structures mentioned in the memory module
- Notes:
    - Will reset and change the values in the segmentData struct as it executes instructions

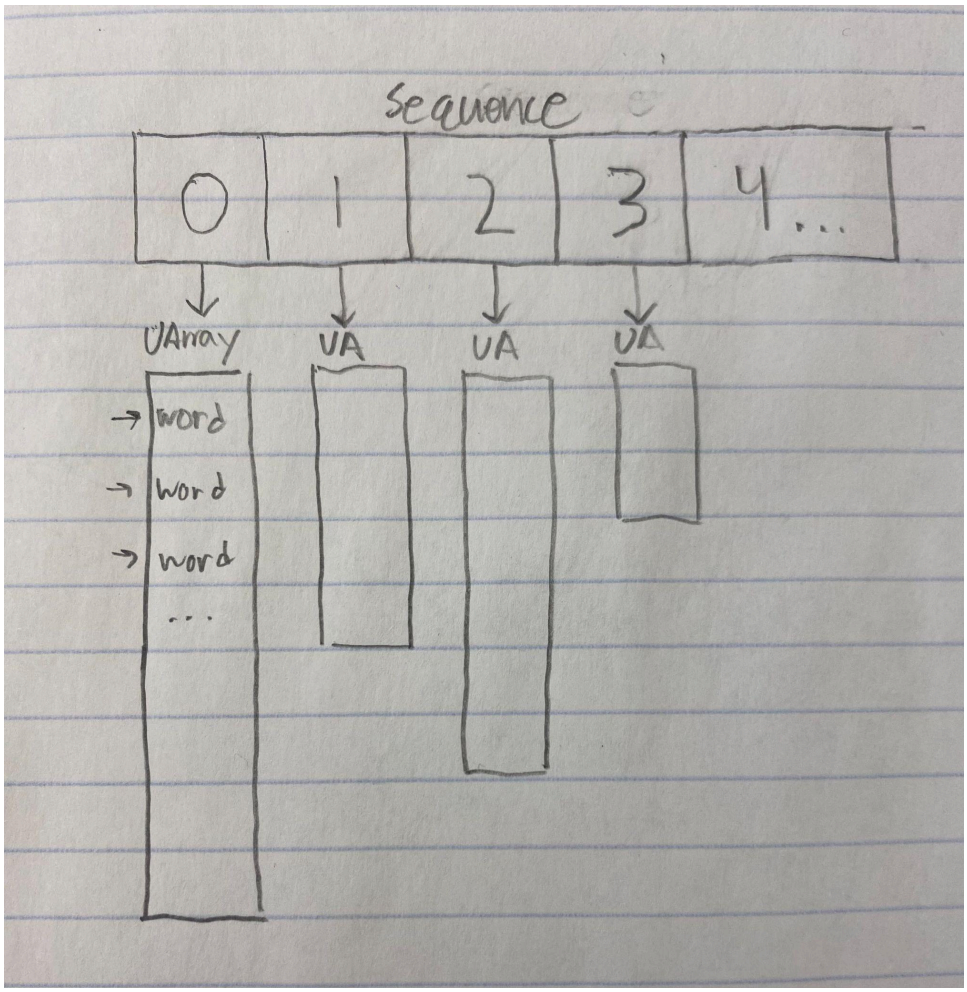
---

Relevant Structure

- Description:
  - We will be using a struct to pass back from our initialization and pass into the function containing our command loop
  - This struct will look like:
  -
```
typedef struct segmentData {
        Seq_T segmentList;
        List_T unusedIndexes;
        int currSegment;
        int currWord;
        UArray_T registers;
} segmentData;
```
- Contained values:
  - segmentList: the sequence with a uarray at each index, which represents an individual segment
    - So this is the list of segments
  - unusedIndexes: this list will contain all indexes present in the sequence that are set to NULL (i.e. segments previously mapped and now unmapped so nothing is there)
  - currSegment and currWord: this will be used to indicate which segment and instruction it should execute next
  - Registers: the uarray will contain the value of each register, with the register itself being the index

---

Diagram of our Structure



Implementation Plan & Testing

- Step 1: Command Line File Handling (main)
    - Implement handling for both reading from stdin and file input
    - Testing:
        - Test with incorrect number of args
            - Expected output: message to stderr
        - Test with non existent file
            - Expected output: EXIT_FAILURE

        - Sanity check: ensure that number of bytes read from stat is correct
- Step 2: IO + Initialize Memory
    - Initially, use stat struct to get all of the bytes in the file and allocate that much memory into the uarray at segment 0
        - Allocation of memory and initialization of the Sequence of uarrays, which is just a list of segments that can be resized accordingly
    - Create the instance of our segmentData struct, setting all variables to there starting values before instructions start to be read
    - We use getc to read in each byte, and continuously update a 32 int “word” using Bitpack_newu
    - Then input that word into the uarray
    - Testing:
        - Sanity check: output all information in a file using fgetc and putchar
            - Expected output: the file contents
        - Test that memory sequence is not NULL
            - Expected output: passed assertion that sequence is not null
        - Test that memory for the first UArray for segment 0 is allocated
            - Expected output: UArray doesn’t return null
- Step 3: Rest of Memory interface
    - Implement the functionality for mapping and unmapping segments
    - How mapping will be done:
        - If a new segment were to be mapped, we initialize a new UArray with the size of register C, and add it to the lowest unused segment in the sequence (which is stored in the struct segmentData)
            - If every index in the sequence is mapped (which will be checked using the length of the list in segmentedData), we will use seq_addhi to add it to the back of the sequence

        - The returned id for each segment will be the index that we store it at
    - How unmapping will be done:
        - Free the uarray at the index, which is the exact same as the id
        - Then set the value at that index in the sequence to NULL
        - Finally, add that index to the list containing all of the unused indexes in the sequence
    - Load program will rely on both of these to run
    - Testing:
        - Write a unit test that maps and unmaps a segment then runs sload or sstore on the recently unmapped segment
            - Expected output: unchecked runtime error
        - Write a unit test that just maps a segment then runs sload to put a value there
            - Expected output: value stored in the newly mapped segment
        - Run Valgrind and memory checks to make sure everything is being freed
            - Expected output: passed valgrind
- Step 4: Instructions Interface
    - This will implement each instruction in the file
    - Each operator code command will be its own function (with helper functions as needed) to execute the command
    - Note: map segment, unmap segment, and load program will take place in memory, but will be called by a simple function in instructions
    - Testing
        - First, run unit tests on some of the more basic instructions (addition, multiplication, division, NAND)
            - Expected output: result of equation
        - Next, run unit tests on a few more (cmov, halt)
            - Expected output: registers being moved accordingly, or the program is halted

        - Then, test input, output, and loadval
            - Expected output: the value loaded into the register
        - Next, test map, unmap,
            - Expected output: proper accessibility to memory at the id (either possible or impossible if it is mapped or unmapped)
        - Lastly, test sstore, sload, and load program
            - Like map and unmap, all of these will take place in the memory interface
            - Expected output: check registers to ensure that they are pointing to the correct segment and offset in that segment; expect no unchecked runtime exceptions
- Step 5: Command Loop
    - This will be one function that receives a pointer to the struct segmentData as an argument and contains a while loop to continue executing as long as there are values to execute and halt is not called
    - Exclusively calls instructions interface
    - Implement a helper function that unpacks the instruction to be called at the start of each loop
    - Testing:
        - After adding a few instructions into segment 0, run unit tests to make sure that the instructions are called with the right registers
            - Expected output: whatever instruction is called, i.e. add would print the sum of the values at registers A and B
        - (these tests would be very similar to those in instructions)
        - Test to ensure that the bitpack returns the correct segment of the “word” that we want
            - Expected output: every register will be a number between 0 and 7, and the rest of the bits do not overload