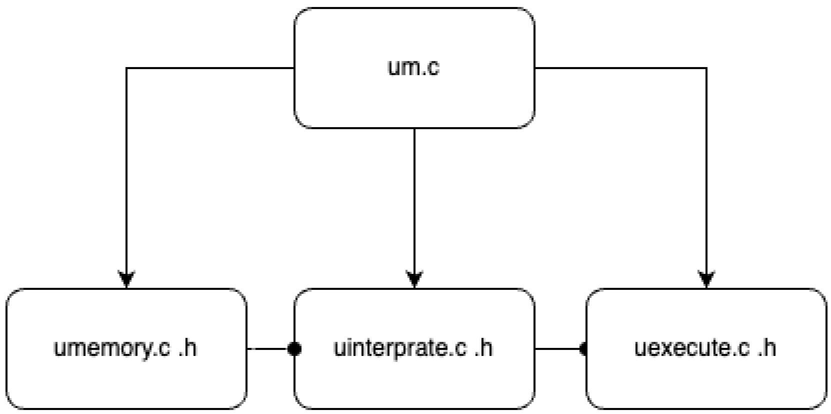
CS40 HW6 Design Doc - Sam Hu (khu04) David Chen (zchen18)

Architecture

Our program will consist of a main driver program um.c and three subsequent modules: umemory, uinterpret, and uexecute. The following diagram shows a basic mapping of the architecture layout:



Upon execution of the program, the contents of the .um program is stored into structures constructed by umemory.c; each line is then parsed by uinterprate.c, which converts the word into actual instructions to execute by uexecute.c (which uses function in umemory to retrieve information from the memory)

Umemory

- General Purpose: Initialize and control a segmented virtual memory system that is represented by a Hanson sequence of Hanson sequences. The memory segment object Mem_T will have three components, listed as follow:
    - A Hanson sequence of Uarray_T: this is used to store the actual contents loaded to memory by the registers; the inner sequence will store uint32_t objects
    - A Hanson sequence of uintptr_t objects (stored as void pointer) to store the segment ids of unmapped memory segments; this component will be used for efficient recycling of used segments
    - A maxID to represent the id of the right most memory segment; this is for convenient access when mapping a new segment.
- List of Functions:
    - Umemory_init:
        - Description: Initialize a Mem_T object and populate m[0] through the inputted program instructions

        - Input parameters: Uarray_T of uint32_Ts that stores program instructions
        - Return parameters: a Mem_T object, with m[0] substantiated
        - Expectation: CRE will occur if the initialization of Hanson sequences failed, which would be handled by Hanson's exception mechanisms
    - Umemory_map:
        - Input parameters:
            - uint32_T size: number of words in the segment
            - Mem_T seg_mem: the current memory structure
        - Output:
            - Uint32_t cur_id: the id of the current segemt
        - Initialize a Uarray_T segment of size "size", set every word value to 0.
        - If there are no unmapped ids, we append the new segment to the end of the segment sequence; cur_id would be max_id + 1;
        - If there are unmapped ids, we remove the first id in the unmapped sequence and set that as cur_id, and use seq_put to place the Uarray_T segment at that location.
        - Expectation:
            - Seg_mem cannot be NULL, otherwise CRE
            - Size have to be bigger than 0
    - Umemory_unmap:
        - Description:
            - We would set the Uarray_T object stored at index on the hanson sequence to be null
            - Append the index onto the end of the sequence of unmapped ids
        - Input parameters:
            - Mem_T seg_mem: the current memory structure
            - uint32_T index: the index of the segment to be unmapped
        - Output:
            - Void
        - Expectation:
            - Seg fault if the given index is currently not been mapped
            - CRE if Mem_T is null
    - Umemory_store:
        - Description: store the inputted value into the designated memory address
        - Input parameters:
            - Mem_T seg_mem: the current memory structure
            - uint32_T mem_id
            - uint32_T mem_offset
            - uint32_T value
        - Output:
            - none
        - Expectation:
            - CRE if Mem_T is null
    - Umemory_load:

- Description: return the value stored at the designated memory
    - Input parameters:
        - Mem_T seg_mem: the current memory structure
        - uint32_T mem_id
        - uint32_T mem_offset
    - Output:
        - uint32_T value
    - Expectation:
        - Fail if returned value is greater than 255
        - Fail if the memory has not being mapped or mapping to m[0]
- Umemory_loadprogram:
    - Description: return the program instructions stored at $m[mem_id], and return the uint32_T pointer that points to the offset; if mem_id is 0, simply return the pointer at the current progran's offset instruction
    - Input parameters
        - uint32_T mem_id
        - uint32_T mem_offset
        - Mem_T seg_mem
    - Output:
        - Uarray_T iinstruction
    - Expectation:
        - Seg_mem is not null
- Umemory_free:
    - Description: free the allocated Mem_T object
    -

Uinterprate

- General purpose: parse a single line of code, split it into operable segments for execution.
- Functions:
    - Getopcode:
        - retrieves the opcode
        - Args: uint32_t instruction
        - Return: the opcode as int
        - Expects the opcode to be in range [0,13]
        - Note: CRE if out of range
    - setReg:
        - Takes a non-loadval instruction and set the corresponding registers
        - Args: the instruction as uint 32, register pointers (uint32 pointers) by reference
        - Return: none
        - Expects: instruction has valid opcode [0,12]

        - Note: CRE if out of range
    - setLoad:
        - Takes a loadval instruction and set the corresponding register and values
        - Args: the instruction as uint 32, register pointer (uint32 pointers) by reference
        - Return: uint32 value
        - Expects: instruction has valid opcode 13, register pointer not null
        - Note: CRE if out of range or register null

Uexecute

- General purpose: execution of the 14 instructions
- Functions:
    - move:
        - Conditional move (if $r[C] != 0 then $r[A] := $r[B])
        - Args: ra,rb,rc by reference
        - Return: none
        - Expects: all three register pointers to be not null
        - Note: CRE if one of the register is null
    - segL:
        - Segmented load ($r[A] := $m[$r[B]][$r[C]])
        - Args: ra,rb,rc by reference
        - Return: none
        - Expects: all three register pointers to be not null
        - Note: CRE if one of the register is null, uses
    - segS:
        - Segmented store ($m[$r[A]][$r[B]] := $r[C])
        - Args: ra,rb,rc by reference
        - Return: none
        - Expects: all three register pointers to be not null
        - Note: CRE if one of the register is null
    - add:
        - Addition ($r[A] := ($r[B] + $r[C]) mod 2^32)
        - Args: ra,rb,rc by reference
        - Return: none
        - Expects: all three register pointers to be not null
        - Note: CRE if one of the register is null
    - mult:
        - Multiplication ($r[A] := ($r[B] × $r[C]) mod 2^32)
        - Args: ra,rb,rc by reference
        - Return: none
        - Expects: all three register pointers to be not null
        - Note: CRE if one of the register is null
    - div:

        - Integer Division ($\$r[A] := \lfloor \$r[B] \div \$r[C] \rfloor$)
        - Args: ra,rb,rc by reference
        - Return: none
        - Expects: all three register pointers to be not null
        - Note: CRE if one of the register is null
    - nand:
        - Bitwise NAND $\$r[A] := \neg(\$r[B] \wedge \$r[C])$
        - Args: ra,rb,rc by reference
        - Return: none
        - Expects: all three register pointers to be not null
        - Note: CRE if one of the register is null
    - halt:
        - Halts the program - reset mr to initial state and then exit
        - Args: none
        - Return: none
        - Expects: none
        - Note: CRE if one of the register is null
    - map:
        - Map the segment, set rb to identifier
        - Args: rb and rc by reference
        - Return: none
        - Expects: both register pointers to be not null
        - Note: CRE if one of the register is null
    - unmap:
        - Unmap a mapped segment
        - Args: rc by reference
        - Return: none
        - Expects: rc not null and stores a valid mapped segment identifier that is not $m0
        - Note: CRE if the register is null, segfault if segment cannot be unmapped
    - out:
        - Output value from rc, must be value [0,255]
        - Args: rc by reference
        - Return: none
        - Expects: rc not null, output value in range
        - Note: CRE the register is null, segfault if output value greater than 255
    - in:
        - Take input and store in rc, must be value [0,255]
        - Args: rc by reference
        - Return: none
        - Expects: rc not null, input value in range (ensured by fgetc)
        - Note: CRE if the register is null
    - loadP:
        - Load a program from memory segment and jump to designated position

        - Args: rb rc by reference
        - Return: none
        - Expects: both register pointers to be not null
        - Note: CRE if the register is null
    - loadVal
        - Load a value into rc
        - Args: rc by reverence and the value
        - Return: none
        - Expects: rc not null, input value in range (ensured by fgetc)
        - Note: CRE if the register is null

Implementation & Testing

The general understanding is that the majority of unit testing comes after the vast majority of memory.c and uinterpret.c is implemented, as .um can only be run then. Most testing prior to that stage will be done using print statements. **The goal is to make unit test usable for some functions. More complicated implementations that require segment operations are implemented after those are tested.**

1. Create the driver function, um.c and the makefile. Build .h files for each module and link them to um.c
    a. Test:: Make dummy print functions in module to test compilation. Should be able to compile and print corresponding statements. Should not have valgrind errors.
2. Implement read_words(): a function in um.c that processes input from the .um file and creates a Uarray_T object where each element is a single instruction in the form of a uint32_T
    a. Test: Print out instructions as uint32, check with commands in unit test compiler for number of instructions & correctness. Should not have valgrind errors.
3. In the umemory module, implement struct Mem_T and umemory_init and umemory_free, implement the argument handler in main. Connect the two. (30 min)
    a. Test: call umemory_init with the created instruction Uarray_T from last step as its input; check whether the first segment in memory (or the Hanson sequence) is valid by using a for loop to repetitively print out each instruction stored in this segment
    b. Test: call umemory_free to free Mem_T object and check for valgrind errors.
4. In the umemory module, implement umemory_map and umemory_unmap; perform combined tests:
    a. Test: call umemory_map repetitively; check whether the returned indices start with 0 and are consecutive 32 bit integers;
    b. Test: cal umemory_map with positive sizes; check whether a correctly sized Uarray_T occurred at the end of the sequence.
    c. Test: call umemory_unmap with a valid segment id; check whether the segment at the given id is erased to null and check whether that index appeared on the unmapped sequence

    d. Test: cal umemory_map repetitively, then call umemory_unmap once to unmap a memory segment that is in the middle; cal umemory_map again a check whether the newly mapped index is identical to the recently unmapped one.
    e. Test: no Valgrind leaks and errors
5. In the umemory module, implement umemory_load, umemory_store and umemory_loadprogram; perform combined tests:
    a. Test: perform umemory_store with distinct uint_32 inputs; retrieve it by getting the uarray stored at the segment index and printing the uint32 stored at the designated offset
    b. Test: perform umemory_load after the previous test; make sure the retrieved instructions are identical with the ones stored in the sequence
    c. Test: perform umemory_store once at a designated position and use umemory_loadprogram to load into m[0]; check if the initial memory position is correctly altered
    d. Test: No Valgrind leaks and errors

6. In main, build the execution cycle and hence set the program counter pointer. Implement constants (opcodes) and registers. Initialize registers by setting them 0.
    a. Test: Check against 3.3 using print to ensure that programs are initialized properly. Should not have valgrind errors.
    b. Test: Print constants and initialized registers. Should reflect corresponding values ([0,13] and all 0); should not have valgrind errors.
7. Build uinterpret.c, connect functions in main so that getopcode is used on the current instruction (indicated by the pointer)
    a. Test: Check valid and invalid instructions (out of range), should call setreg/ set load with correct output values or CRE
    b. Test: No Valgrind leaks or errors
8. Build halt, which frees all memory segments when called and then calls exit to terminate the run.(15 min)
    a. Test: Check with halt.um and valgrind. Should run successfully without error/leak.
9. Build simple execution functions that does not require map/ load program. Build and test output first so print statements are no longer needed
    a. Test: Test output with readable ascii, non readable ascii, and values out of range. Should be able to print in console, pipped to files, and cause segfault, respectively.
    b. Test: Each other operation should be tested with unit test thoroughly, for regular cases, as well as edge cases(values that trigger the mod 2^32, for example) and null register inputs. Should all behave as per the function contracts.
10. Implement map functions in umemory.c and uexecute.c.
    a. Test: Test with unit test operations for regular cases as well as cases specified in 3.3. The former should run smoothly and the latter should cause segfault
11. Implement seg load/ store functions in umemory.c and uexecute.c.
    a. Test: Test with unit test operations for regular cases as well as cases specified in 3.3. The former should run smoothly and the latter should cause segfault

12. Implement load program functions in umemory.c and uexecute.c. There should be a special case to handle cases where rb = 0
    a. Test: unit test case where another segment is loaded
    b. Test: case where a jump occurs, should skip outputs (other operations too but this is observable) between jump
    c. Test: case where a infinite loop occurs, should induce timeout
    d. Test: case where rb or rc is out of range, should result in segfault

Test prototypes (does not include.um)

Code chunk for step 3:

```
typedef struct Mem_T {
       Seq_T seg_mem;
       Seq_T unmapped;
       uint32_t maxID;
} *Mem_T

const char* FILETOLOAD = "test.um";
const uint32_t INST[4] = [0x1f2b7444,0x1f2b7444,0x1f2b7444,0x1f2b7444]

FILE *fp = fopen(FILETOLOAD);
UArray_T prog = read_words(fp);/*assuming this is correct */
Mem_T memory = mem_init(prog); /*function tested*/

UArray_T testArr = Seq_get(memory->seg_mem);

for(int i = 0; i < Uarray_length(testArr);i++){ /*compare each instruction with
standard answer*/
       uint32_t num = (uint32_t)(uintptr_t)Uarray_at(testArr,i);
       printf("the result %s for instruction %d \n", ((num == inst[i]) ? "MATCHED!" :
"DOES NOT MATCH!"),i);
}

fclose(fp);
umemory_free(&memory);/*function tested*/
```

Code chunk for step 4:

```
/* Tests the basic functionalities of map and unmap, will require more tests since
this is just a prototype */
```

```
void umemory_map_unmap_test(Mem_T mem)
{
        /* check whether mapping returns consecutive ids and allocated correct space*/
        uint32_t id;
        uint32_t size = 64; /* random number to test with */
        for (int i = 0; i < 1000; i++) {
                id = Umemory_map(mem, size);
                assert(id == i + 1);
                assert(UArray_size(Seq_get(mem->seg_mem, id)) == size);
        }
        /* call unmap with an invalid id */
        uint32_t unmap_id = 114514;
        Umemory_unmap(mem, unmap_id); /* Should fail the program */
        /* unmap a segment in the middle and call map again, check whether the two
segment indices are the same */
        for (int i = 250; i < 500; i++) {
                unmap_id = i;
                Umemory_unmap(mem, unmap_id);
                id = Umemory_map(mem, size);
                assert(unmap_id == id);
        }
}
```

Code chunk for step 5:

```
/* Tests the basic functionalities of load and store, will require more tests since
this is just a prototype */
void umemory_load_store_test(Mem_T mem)
{
        uint32_t size = 64; /* random number to test with */
        UArray_T cur_program = Seq_get(mem->seg_mem, 0);
        /* assuming 10 consecutive slots in memory has been mapped */

        for (uint32_t i = 1; i <= 10; i++) {
                for (uint32_t j = 0; j < size; j++) {
                        /* test whether value stored by umemory_store can be fetched by
umemory_load independently */
                        uint32_t random = generate_random_uint32(); /* some helper
function that generates a random uint32 value */
```

```
                        Umemory_store(mem, i, j, random);
                        uint32_t load_value = Umemory_load(mem, i, j);
                        assert(load_value == random);

                        /* test whether Umemory_loadprogram successfully replaces the
program Uarray */
                        UArray_T temp = Umemory_loadprogram(mem, i, j);
                        assert(temp != cur_program);
                }

        }



}
```