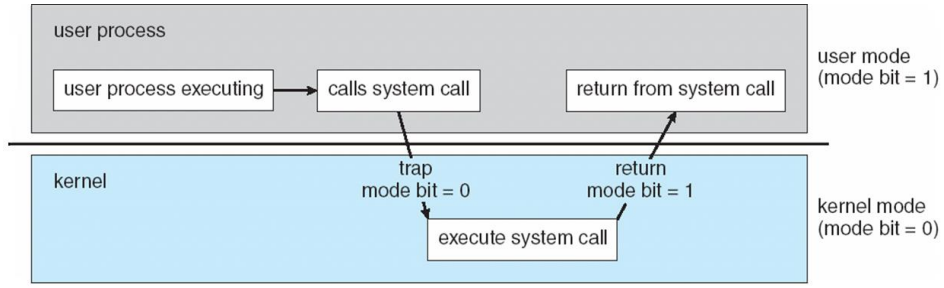
Build Your Own Virtual Machine

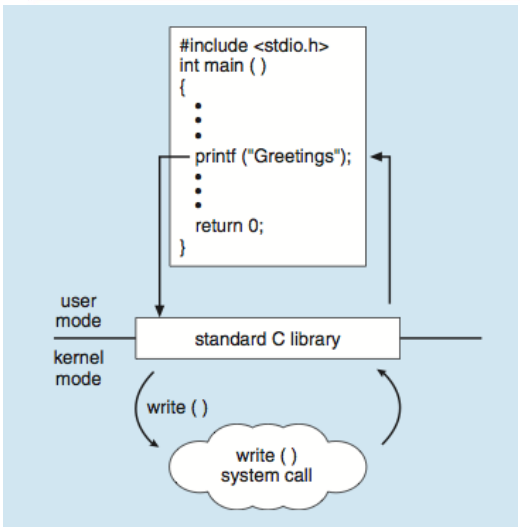
Objective:
Implement a Virtual Machine with simulated CPU, main memory, hard disk and necessary components. Implement a loader that can load osX assembly byte code into the memory. Implement an interpreter that can execute the fetch-decode-execute cycle. You can successful execute given osX assembly code on you Virtual Machine.

Synopsis:
Run your Virtual Machine
VM-> shell change to MYVM->
load –v program.osx
run –v program.osx
coredump –v program.osx
errordump –v program.osx

Hints: Build all the OS services as module. Provide system programs to call those modules and this is your Shell with System program. Thinking a about using a switch statement?

Related Knowledge
(1) Dual-Mode





(2)The Assembler

- A Two-Pass osX Assembler (osx.exe) has already been implemented for you.
  Usage:
  osx <osx .asm file> <loader address> [-v]
  <osx .asm file> valid osX assembly file
  <loader address> address where loader is to load program
  -v option turn on verbose mode\n"
- If you want write out own osX assembler:
  - In the first pass load a Symbol Table (Associative Map) as you parse through the assembly code.
  - The key of the Symbol Table is a Label (Code, Data) from an assembly program. The value associated with the key is the location of the Label (Label → Address). Compute the address of the Label during the first pass.
  - In the second pass create byte code for your program. Using the Symbol Table convert Labels to addresses. Your byte code must be all numeric data: (e.g., ADD is 16, R7 is 7, Label A is 1024, etc.)
  - To parse instruction in the first and second pass read one (1) line of data at a time from your assembly file.
- Example of osX assembly code:
  Random example of osX assembly code instructions. Any instruction can be preceded by a label:

```
COUNT  .INT  1
INDEX  .INT  0
NEXT   .INT  5
R      .BYT 'R'
A      .BYT 'A'
Y      .BYT 'Y'
       LDR R0, COUNT
       LDR R1, INDEX
       ADD R1, R0
       STR R1, INDEX
       TRP 1
BEGIN  MOV R3, R0
       CMP R3, R5
       BNZ R3, BADX
       TRP 3
       TRP 0
```

  For more details see Appendix.

Details:

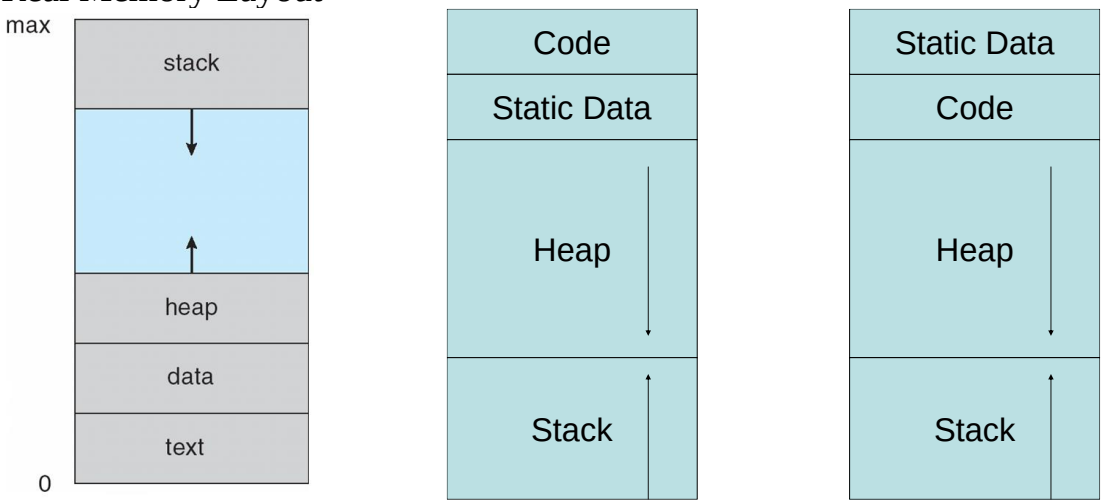
(1) Write a Virtual Machine (VM) that can execute your osX byte code. You will need to load the bytecode (.osx file) into memory before you can begin to execute code.

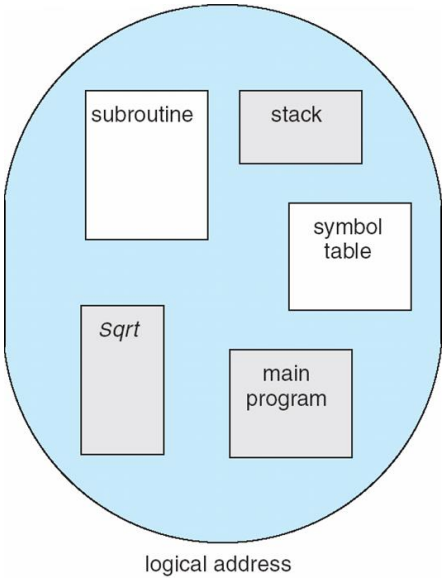
- Hardware: CPU, register, memory, clock, cash, etc. Add hardware as needed as the project evolves.
  You need to decide the Memory Layout first. There are a few major options to consider. Pick one solution or the other. Don't Mix them together that is not a

solution. That is a MESS! Just make a choice a stick to it! Don't waffle that just wastes time.
• Starting with Code first makes it easy to determine your initial PC (PC=0).
• Starting with Static Data first seems to be more readable than the inverse.

- Data structure: array, stack, heap, etc.

Real Memory Layout





(2) The Loader
Implement a loader to load your program into the main memory according to memory model you picked. You can use switch-statement.

Example:

```
switch(byte) {
            case CMP:
            case MOV:
```

```
                    printf("%i: Loading %s (%i)", i, INSTR_NAME[byte],
byte);
                    fread(&byte, 1, 1, ifile);
                    printf(" %s (%i)", REG_NAME[byte], byte);
                    fread(&byte, 1, 1, ifile);
                    printf(" %s(%i)\n", REG_NAME[byte], byte);
                    fread(&byte, 1, 1, ifile);
                    fread(&byte, 1, 1, ifile);
                    fread(&byte, 1, 1, ifile);
                    break;
              case STR:
              case STRB:
              case LDR:
              default:
                    break;
```

(3) The Interpreter/Instruction Cycle
Simulate the basic fetch-decode-execute instruction cycle includes fetch the instruction, decode the instruction, fetch operand, execute the instruction. You can use a while-statement and a switch-statement. Increment the Cycle by one for each instruction cycle.

Example:
Registers are stored in the array reg[]; Memory is in the array mem[]. In pseudo code you can have something like the following:

```
PC = Beginning_Address;
Running = True;
while(Running) { // Big Switch
      // fetch the current instruction from memory
      IR = mem.fetch(PC);
      //decode and execute
      switch(IR.opCode) {
      case ADD:
            reg[IR.opd1] = reg[IR.opd2] + reg[IR.opd3];
            PC += 6Bytes;
            break;
      case MOV:
            reg[IR.opd1] = reg[IR.opd2];
            PC += 6Bytes;
            break;
      case LDR:
            reg[IR.opd1] = getMemoryAsInt(mem, reg[IR.opd2]);
            PC = 6Bytes;
            break;
      // Code on the instruction in your architecture

      }
            CLOCK++;
}
```

(4) The osX assembly instruction will give you all the weapons you need ☺.

Interrupts
SWI <imm> ; Software Interrupt

Encode: 1 byte op code | 4 byte immediate | 1 byte unused
SWI = 20
Suggested Usage: Execute interrupt <imm>

Software Engineering Approach

(1) Software Design
Architecture Diagram, Class Diagram

(2) Testing
Design Test Cases:

i) asm files
ii) inputs
iii) expected outputs
iv) write a test case doc separate with readme.

Thoughts:
(1) How do you design and implement a Shell for your VM?
(2) How is your OS kernel loaded in to your VM in your design and implementation?
(3) How do you implement user mode and kernel mode?
(4) Should you chose a layered approach, modules approach, microkernel approach, or hybrid approach? Which one do you choose? How do you design and implement it? This is critical in architecture design, and is one of the critical factors that determines whether your project will success or fail.

Acceptance Testing

1	shell	5
2	load –v program.osx, memory model	20
3	run –v program.osx, fetch-decode-execution	20
4	coredump –v program.osx	10
5	errordump –v program.osx	5
5	read me, engineering glossary list	10
7	Architecture diagram (system, memory), class diagram	10
8	Test cases	10
9	User mode and kernel mode design and implementation	10