CS 4380 Project 4

Build upon Project 3, Implement a Virtual Machine, which can execute the following Test Program.

Submit your project via Canvas (YourName_p4.zip) with your source and executable

- **README.md file with**
  - **any dependencies** *(including how to install/configure them)*
  - **how to build it** *(if C++ for example, a working make file)*
  - **if you want me to regrade a previous project's submission, using the source code from Project 4.**
- **A batch file or .exe to use for running it**
- **Project Source files (entire project/solution zipped recursively)**
  - **please put your main function in a source file called main, such as main.py, main.c, main.cs, main.cpp, main.java etc. in the *root* of the project's folders**
  - **asm\proj4.asm** *(should **only** contain parts 1-2, the **single-threaded**, recursive Fibonacci [Part 1] and Addressing Mode Test [Part 2])*
  - **asm\proj4_ec.asm** *(if doing extra credit [such as multithreaded, a coalescing heap, or something hardware specific] you should put the extra credit assembly code in asm\proj4_ec.asm).* Remember: (a) pass off parts 1-2 and (b) get instructor approval (c) ***before*** doing any extra credit work.
  - **asm\proj1.asm, asm\proj2.asm, asm\proj3.asm** (and a comment in the README.md file) if you want me to regrade them those projects, using your latest, VM/Assembler and the latest .asm files.
  - *you may have more .asm files than these, but only asm\proj*.asm files will be graded (others will be ignored)*
  - **asm\proj4.bin** (generated from your proj4.asm file)
  - **if you're doing extra credit: asm\proj4_ec.bin** (generated from your proj4_ec.bin)

Implement the following.
NOTE: You must implement parts 1 and 2 below you can't JUST embedded them in part 3 only and expect to be given credit for parts 1 and 2. Part 1 and 2 are single threaded while Part 3 uses Parts 1 and 2 and is multi-threaded.

Part 1 and 2 are sequential code, not parallel!

Part 1, REQUIRED: Recursive Fibonacci Function Test

Create an array (named ARR) with space for 30 integer values.

Implement a recursive Fibonacci function by using a run-time stack to both pass parameters and return a result. You must place an activation record on a run-time stack! Repeatedly compute and printout the Fibonacci value for an integer entered from the keyboard. The key to stop computing values is when the value ***-1*** is entered, but if the array ARR fills up, you must also stop processing additional input.

You can't use caching. You can't use iteration. It must be implemented using recursion only. Output MUST look like this (where <n> is the number entered via TRP #2, and <fib(n)> is the computed Fibonacci value):
:

Fibonacci of <n> is <fib(n)>

Each time an integer is entered for the recursive Fibonacci function and fib(n) is calculated, also place the n and fib(n) values into the array (start at the beginning, ARR[0]). ARR = [n_1, fib(n_1), n_2, fib(n_2), n_3, fib(n_3), …, $n_{i \le 15}$,fib($n_{i \le 15}$)]; since 30/2 = 15).

When the value of -1 is entered ***or*** if 15 sequences have been processed (the ARR buffer is full), go to Part 2 below.

Part 2, REQUIRED: Addressing Mode Test

Print the integers that were entered into the array starting at both ends and meeting in the middle.

- Print the first and the last elements in the array,
- Then print the second and second to last elements in the array
- Then print the third and third to last elements in the array
- Etc.,
- Until you've printed the middle two numbers.
- **Please don't re-ask for array values! Use the n and fib(n) values from part #1 otherwise, you'll lose almost all of the points for part #2.**

Example of Part 1 and Part 2:

Part 1:

int ARR[30] = { 0 };

<enter 6>, fib(6) = 8, therefore, print "Fibonacci of 6 is 8",
Append both numbers (n and fib(n)) to the array of numbers
ARR[0] = 6
ARR[1] = 8
ARR = [6, 8]

<enter 3>, fib(3) = 2, print " Fibonacci of 3 is 2" and append both numbers to the array
ARR[3] = 3
ARR[4] = 2
ARR = [6, 8, 3, 2]

<enter 7>, fib(7) = 13, print " Fibonacci of 7 is 13" and append both numbers to the array
ARR = [6, 8, 3, 2, 7, 13]

<enter 4>, fib(4) = 3, print " Fibonacci of 4 is 3" and append both numbers to the array
ARR = [6, 8, 3, 2, 7, 13, 4, 3]

if -1 entered or after 15 times (when the array is full) stop taking input (go to **Part 2**).
<entered -1>
ARR still equals [6, 8, 3, 2, 7, 13, 4, 3]
goto Part 2

Part 2:

Using the array print the first and last, print second and second to last, print third and third to last, ..., etc. until you reach the middle of the array (in the example above [2, 7] are the two numbers in the middle of the array):

6, 3, 8, 4, 3, 13, 2, 7

Part 3, ***Extra Credit*** Multi-Threaded VM (up to 5 points)

Students who fully understand function calls from Project 3 indicate implementing this is not very difficult.

Implement a multi-Threading architecture for your VM using the following instructions. Note, this is a simplified multi-Threading (not complete) architecture.

- RUN R_{dest}, LABEL
  Run signals to the VM to ***create a new thread***.

  REG will return a unique thread identifier (number) that will be associated with the thread. The register is to be set by your VM, not by the programmer calling the RUN instruction. You can determine what action to perform if all available worker threads are in use (preferred, NOP [stall] in the main UI thread, but throwing an exception is fine).

  Running out of identifiers means you have created too many threads and are out of the maximum number of thread stacks that were allocated. However, once a worker thread finishes, it can be treated as a thread pool and used for a new request.

  The LABEL will be jumped to by the newly created thread. The current thread will continue execution at the statement following the RUN.

- END
  End will **terminate** the execution of a **non-MAIN** (worker) thread.

  In functionality END is very similar to TRP 0, but only for worker threads. END should only be used for worker threads.
  - END will have no effect (or should raise an exception), if called in the main UI thread.
  - Similarly, TRP 0 has no effect on worker threads (or should raise an exception) if called on a worker thread.

- BLK
  Block will cause the MAIN thread (the initial thread created when you start executing your program) to ***wait for all other threads to terminate (END)*** before continuing to the next instruction following the block. It's basically like a WAIT_ALL or JOIN OS call.
  - Block is only to be used on the MAIN thread.
  - BLK will have no effect (or should throw an exception) if called in a Thread which is not the MAIN thread.

- LCK LABEL
  Lock will be used to implement a blocking critical section. Calling lock will cause a Thread to try to place a lock on the mutex identified by Label.
  - If the critical section is already entered (locked) by another thread, then LCK-ing thread will block (NOP) until the critical section is exited (unlocked).
  - The data type for the Label is .INT
- ULK LABEL
  Unlock will remove the lock from a mutex. Unlock should have no effect on a mutex if the Thread did not lock the mutex (a.k.a., only the Thread that locks a mutex should unlock the mutex).

Once you implemented the new instructions create the following test code to clearly show that your multi-threading architecture works by reusing the code from parts 1 and 2 above to call your Fibonacci function on multiple threads (you must have 4 worker threads, plus the main thread). ***See Project4_PreemptiveMultitasking.txt file (in the Files | Projects folder in Canvas) for Part 3's pseudo-code.***

Hints:

1. Ensure that in addition to the individual stacks created for all fivethreads (main + 4 workers), to also create a thread context for each one. This can either be two separate places in memory (like the bytecode and registers are) or, like the slides suggest, combine each thread's stack and context into one contiguous data structure (block of memory).
2. When the VM launches, in addition to the bytecode and stack for the main thread, add an area of memory (the slides recommend after the stack) for main thread's context.
3. Calculate and allocate, all five thread's (one main UI thread + four worker threads) stack and context.
4. Set the SB, SL, SP, and FP for each thread ***IN*** each thread's context when the program begins or when RUN is invoked (the advantage of when RUN is invoked, allows you to reset SP and FP each run.
5. Set the PC ***IN*** the thread context for that thread, aka, the offset to the label that's part of the run command
6. Don't forget to update a thread context at the end of that thread's context swap.
7. Use locks and unlocks to access the common Array in step 2.
8. You don't need to **rewrite** the Fibonacci function just call it on multiple threads.
9. You will need to **rewrite the code that invokes the Fibonacci function**. The diagram below gives you an outline for the multiple threaded version.
10. I expect to see interleaving when printing "Fibonacci of <n> is <fib(n)>"
11. Interleaving means that the print out of "Fibonacci of <n> is <fib(n)> of X will be interleaved. (This is a race condition.)
12. We test this by entering X values like 1, 7, 11, 15 and should get little to no interleaving, while X values like 7, 6, 7, 6, 7 should produce lots of interleaving.
13. It's basically like a WAIT_ALL or JOIN OS call.

Part 4, Extra Credit Option Porting Part 1 and Part 2

Students can earn 7 points extra credit by porting Part 1 (recursive Fibonacci function) and Part 2 (array processing) to any alternative architecture (use a simulator, emulator, or actual hardware). I will not accept a port to the hardware used in the CE courses. I want you to learn something NEW or at least show you have learned something on your own. Your port must work exactly as defined in part 1 and part 2 to get ANY credit **(½ way working ports don't get ½ credit).**

Example architectures are the MIPS (SPIM simulator), ARM (Raspberry Pi/QEMU emulator/ARMsim University of Victoria), Arduino (ArduinoSimulator), x86 (MASM). Based upon some research I've done it should be relatively simple for you to port to either MIPS or ARM as they are both examples of RISC-based architectures. The x86 might be a little more effort as it is an example of a CISC based architecture and I've done little looking into the Arduino.

YOU MUST BRING YOU PORT OF PART 1 AND PART 2 TO MY OFFICE AND HAVE ME GRADE IT WITH YOU IN MY OFFICE. I WILL NOT GRADE NOR GIVE YOU CREDIT FOR WORK YOU DON'T EXPLAIN TO **ME IN PERSON!**

***Don't just submit your port via Canvas*!** This means you're welcome to use hardware you own to develop your port just bring it to school and show me what you have done!

Part 5, Create your Own Extra Credit Project

(max of +10 points above Option 4)

1. (up to +5) As an alternative to the above options you can implement multiple new instructions that you create. Then create a program to show how your new instructions work. Focus on creating instructions that provide your VM with some interesting new capability, not just adding a collection of random instructions. ***To be acceptable it has to include (a) either the multithread VM (b) and/or heap, <u>plus</u> something different and new.***
2. (up to +5) Implement a heap (heap management via VM, addressing heap via .asm file.
   a Using a fixed-size block (such as 64 bytes), then use ALLC R_{dest} to allocate the next fixed size data block on the heap. VM's Memory Manager will return address of data block on heap in R_{dest}
   b Note you don't have to implement FREE for this option or for the compiler class either! However, if you do implement FREE, implement it as FREE R_{source} (where Rsource is the address that was returned in R_{dest} using ALLC

3. (up to +7) Implement a variable-sized, coalescing heap
    a ALLC R_{dest}, IMM (the additional parameter, IMM is the number of bytes to allocate on the heap). R_{dest} is where the heap address allocated is returned.
    b In your memory manager, use a circular linked list
        c You MUST implement FREE R_{source} to free items on the heap (R_{source} is the R_{dest} return from ALLC

YOU MUST get approval from me before implementing your own instructions, by having a face-to-face meeting. NO EXCEPTIONS!!! Failure to follow this guideline will result in me not grading your project.

The project is worth 100 points Systematic Deductions are taken off the top before individual Project Deductions are made. Systematic Deductions are like penalties for doing something a Sr. shouldn't do because they are just too fundamental and simple.

Systematic Deductions

Late	-100
Naming executable or asm file incorrectly	-15
Sending an project I have to regrade	-40 (may not get graded) un-executable project
changing syntax/semantic	[I most likely never get around to regrading]
VM hangs when finished	-15
VM crashes during execution	-50 or greater (may not get graded)
Debug Output	-15
Failure to send all the elements in your zip	-15
Intermixing code and data sections	-15 Project

Deductions

Missing part 1 & 2

Blows up during execution	-40
Wrong Fibonacci Value	-30
Multi-threaded Fibonacci without Single Threaded	-25
Missing Single threaded Array Printing	-20
Printing wrong Array values	-15
Re-asking for values between parts 1 & 2	-15

Multi-threaded Recursive Fibonacci Function

Output must be interleaved:

interleaving must change with input values	5
Wrong Multi-Threaded Fibonacci Values	-25
Wrong Multi-Threaded Array Values	-10

Other Hints:

Recursive Fibonacci Function

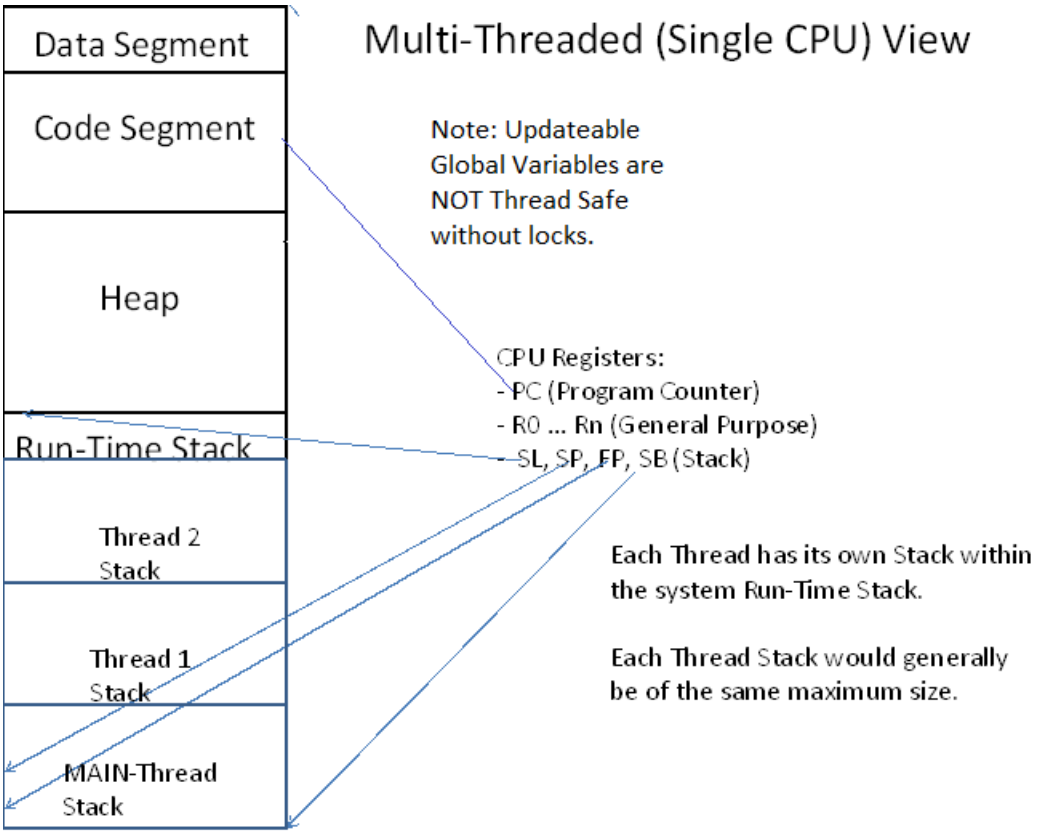
```
int fib(int n) {
    if (n <= 1) return n;
    return fib(n-1) + fib(n-2);
}
```

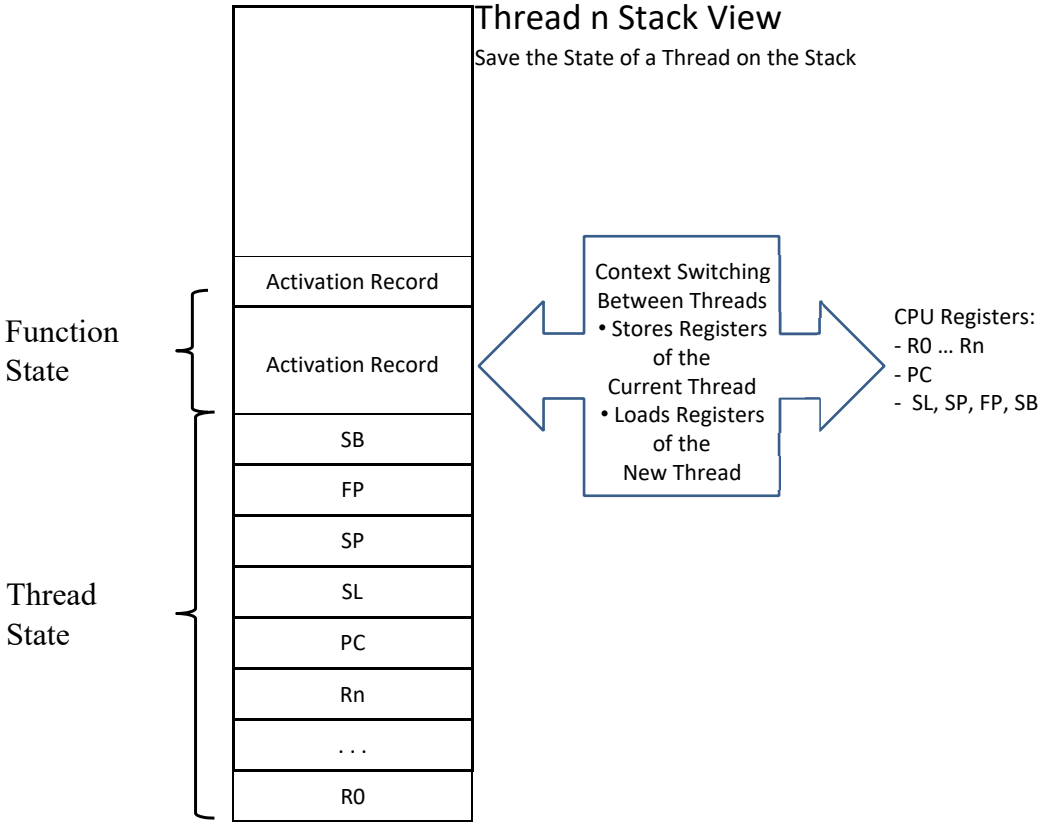
Addressing Mode

You will be dumping the contents of the array working from both ends. Remember the array always has an even number of integers (n and fib(n)) value), but it may not have 30 integers. The following array [5,5,2,1,8,21,12,144] has a CNT of 8 and is printed as follows:
5, 144, 5, 12, 2, 21, 1, 8,

Multi-Threaded VM

- Thread Stacks are of a Fixed Max Size (TS)
- The MAIN Thread always has Thread Id of 0
- To locate a Thread Stack, SB - (TS * Thread Id)
    - Note base address + offset again
- To locate the Stack Limit of a Thread Stack
    - SB – (TS * Thread Id) - TS
- Using Round Robin Scheduling is simple and effective (Execute K instructions per thread, ***keep K at 1 for this program***)
- Printing out the Thread Identifier of a thread as it executes is a ***GREAT WAY*** to show that multi-threading is working (but only if the Thread Identifiers are interlaced), but this is DEBUG output and must be suppressed in what you submit to Canvas.
- Consider using a value of -1 for an unlocked mutex and the Thread Id for a locked mutex.


Multi-Threaded (Single CPU) View
Data Segment
Code Segment
Heap
Run-Time Stack
Thread 2 Stack
Thread 1 Stack
MAIN-Thread Stack
Note: Updateable Global Variables are NOT Thread Safe without locks.
CPU Registers:
- PC (Program Counter)
- R0 ... Rn (General Purpose)
- SL, SP, FP, SB (Stack)
Each Thread has its own Stack within the system Run-Time Stack.
Each Thread Stack would generally be of the same maximum size.


Thread n Stack View
Save the State of a Thread on the Stack
Activation Record
Activation Record
SB
FP
SP
SL
PC
Rn
. . .
R0
Function State
Thread State
Context Switching Between Threads
• Stores Registers of the Current Thread
• Loads Registers of the New Thread
CPU Registers:
- R0 ... Rn
- PC
- SL, SP, FP, SB