CS 232: Operating Systems

Logical Address Support in CalOS

The purpose of this homework is to explore what needs to be done to add support in CalOS for programs being written using logical addresses instead of hard-coded physical addresses.

Current State of the OS

Accept your repo by clicking here, and then cloning the repo, with `git clone <your-repo-url>`

CalOS has changed a bit since homework **ctx_switching**:

- The OS now supports multiple CPUs, although the code I've given you only creates and uses one CPU.
- `current_proc` is now a list of PCBs for the currently running processes, indexed by the CPU number.
- Software interrupts (traps) have been added. Specifically, if a program tries to execute an illegal instruction or when a program ends normally, a trap is generated.
- Debugging output messages have been cleaned up a bit.

The MMU, as you recall from class, is a device that sits between the CPU and RAM. It is responsible for checking if address accesses are legal, and can be used to translate addresses. Using the MMU we can begin to treat addresses in code as logical addresses instead of physical addresses and we can thus load a program into any memory location in RAM.

Step 1: Add the MMU class, but don't have it do any address translation

Create an MMU class in the file **ram.py**

- The constructor takes a reference to the `ram` object and stores it as an instance variable.
- Create a method which the cpu can call to get a value from memory. This method takes an address as a parameter, and (at this point) simply calls the `ram` object to get the value from memory, and returns that value.
- Create a similar method to store a value in memory -- thus, the method takes two parameters, an address and a value. Again, just pass the value through to the `ram` object, which stores the value at the given location.

Now, update **cpu.py** to use the MMU and not use RAM directly.

- Update the CPU constructor to create and store an MMU object, passing in the `ram` object as the parameter.
- Update the code in the CPU class to use the MMU API to get and set values from memory. There should be no direct access to RAM anymore.

Verify that you can still run programs successfully. See homework05 for examples on loading and running **mult.asm** and **fib.asm**.

Step 2: "Fix" mult.asm to use contiguous memory

One problem with our assembly language programs is that they are written with hard-coded physical addresses in them, not only for places to jump to in the code, but also addresses which data is read from and written to.

Update **mult.asm** to use contiguous memory:

- The code in **mult.asm** assumes the data is found in locations 10, 11, and 12, whereas the code is assumed to be at locations 20 - 31.
- Change **mult.asm** to read the two values to multiply together from locations 32 and 33, and put the result of multiplication at location 34. Thus, the code will be from locations 20 to 31, and the data will immediately follow in locations 32 - 34.
- Test your code to see that it still works.
- Those doing the course for honors: do this for **fib.asm** as well.

Step 3: Update main.py and asm files to note the area in a file for data storage

Our assembly language files now store data in the locations following the text of the code. But, the files do not indicate the location or size of the data.

To solve this problem, we are going to make one assumption: code always uses the area right after the text as a data area. That solves the problem of where the data will be stored in memory. However, we still need to indicate to the operating system how large the data area is. If we don't do this, the operating system cannot properly check if memory accesses go out of range. To fix this, we'll introduce another special label in the assembly language, `__data: value`, which will indicate to the OS how much space must be reserved for the program to store data.

- In **main.py**, inspect the code in `_handle_main_label`. Note how the code reads the address of the entry point of the code and stores it in the PCB.
- Add a new method `_handle_data_label` that will take the same parameters as `_handle_main_label`. This code reads the number of bytes from `line` and calls

`pcb.set_high_mem()` to store the high *memory location* in the PCB. You will have to write a tiny bit of code to compute that high memory location.
Update the debugging messages to tell the user the important information that has been read and stored in the PCB.

- Add a `__data` line to the **mult.asm** file with the value 3 to indicate to CalOS that **mult.asm** requires that 3 bytes of data space be reserved after the code. Note: this `__data` line goes at the end of the file -- after the last line of the program.

  Honors students: do this for **fib.asm** as well. Note that the program currently generates output starting at location 500 but is not limited in how many values it can output. You may assume a limit of 100 bytes.
- You need to update `_load_program()` to do two things:
  - Add a line just before `for line in f:` to call `set_low_mem()` and store the `addr` value.
  - Add another `elif` to recognize the `__data:` line and call `_handle_data_label().`
- Run your code now, to make sure everything is still working. Turn on debugging so that you can see if the low and high memory locations are being stored correctly in the PCB. (NOTE: I found it useful to update the `__str__` method in PCB to print out the low and high memory limits for the process.)

We are still not doing any address translation or address checking yet. In fact, our MMU object doesn't do anything yet, except pass memory accesses on through to the RAM object. It is time to take care of that. We are going to have our MMU object implement figure 7.6 from the book, where low and high addresses are checked and addresses are translated from logical to physical.

Step 4: Implement Address Translation and Checking

Add registers to the MMU

- In the MMU constructor, add two instance variables: one is the `reloc` register and the other is the `limit` register. Initialize those values to 0.
- Implement `set_reloc_register()` and `set_limit_register()` so other code can set those values.
- In the next step, you'll implement figure 7.6, which assumes that logical addresses are 0-based. Thus, you'll have to change **mult.asm** to use 0-based addresses. Do that now. (Sorry.) Make sure the value after `__main:` is 0, indicating that the code is written assuming logical addresses starting at 0. Honors students: do it for **fib.asm** as well.
- Implement figure 7.6 in the rest of your code in the MMU class to do address translation and checking. If an address is illegal (too high) just print an error like **"BAD ADDRESS %d: too high"**. The code that tries to get a value out of range will fail sooner or later, even if we don't explicitly halt it for accessing bad memory.

- Update code in **calos.py** to set the mmu registers whenever setting CPU registers. Do this by creating a method in the CPU class called `set_mmu_registers()`. Note that the values passed to `set_mmu_registers()` aren't a direct mapping of PCB values to MMU registers -- you have to do a little math to set the correct values in the mmu.
- Test your code by loading **mult.asm** into a location that is not 20 -- like 30 or 200. You'll have to enter the operands to be multiplied into the locations that are 12 and 13 bytes after where you load your code.
  Those doing the course for honors: do this for **fib.asm** as well.
- Test your code by loading **mult.asm** twice, into two different locations in RAM, setting the operands for both processes, and then starting the OS. Both instances of **mult.asm** should run concurrently and produce correct results.
- Alter **mult.asm** to try to access values outside of the legal range. Make sure the error messages are printed on the console. After doing this, undo your changes so that **mult.asm** runs correctly again.

Checking In

Submit all the code and supporting files for this homework by doing a git add, git commit -m , and then git push, or using VSCode to sync your files to github.com.

We will grade this exercise according to the following criteria: (15 pts total)

- 15 pts: code executes correctly and is hospitable.
- As before, debug print output should **only appear** when debugging is turned on.
- Make sure you use good variable names so that your code is self-documenting.