# Question 1

* + Every time we call fork we create a new process, and run exactly the same program
  + Similar to the example from class
  + Diagram

    Description automatically generated
  + Diagram

    Description automatically generated
  + You can compile the code and test it too
  + WE COPY ALL DATA from the LAST state that the parent process was in! so there won’t be a infinite loop; thus, there is no reinitialization of any variables (such as loops that have already been entered by the parent).

# Question 2

* + NOTE: y = y+1 is not an atomic statement; notice the given information about what is atomic
  + Text

    Description automatically generated
  + Up to **6** different answers possible
  + Add and time = + and \* are not atomic; you have to load, do the operation, and store back to memory. Thus, there are three atomic instructions that make up addition and multiplication operations.
  + Remember when we context switch we save register data to a state block; thus, remember, registers are not shared between threads, but the **public variable memory locations are shared**; thus when we load and store
  + i understand what you mean Mustafa, but here is the catch: if we treated all the instructions in Foo and Bar as atomic, yes they would still create a race condition and the output would be indeterminate; however, since the question says that the assembly-level instructions are not all atomic, then it allows for more combinations of indeterminate outputs. Think about it like this: if you switch between two functions, each with 2 statements, you could get 2 possible outputs, but if you have two functions where each one has lets say 7 functions, you can have more possible indeterminate outputs

# Question 3

* + remember the 4 principles of CS solutions

# Question 4

* + solution should be similar to test and set solution:
    - Graphical user interface, text, application

      Description automatically generated
    - But replace test and set with the new atomic function given and make other adjustments concerning initial values, etc.
  + So imitate the above solution and replace test and set with Inc atomic function given.

# Question 5

* + You could end up with same machine allocated to two different users; we need a solution to solve this.
    - Need to make this section run atomically; can use semaphore:
    - Graphical user interface, text, application, Word

      Description automatically generated
    - Or do it on this part:
    - Graphical user interface, application, Word

      Description automatically generated

# Question 6

* + Need to make sure you get the same program output as when all threads were combined into one program in that specific order to get the output for a and e. SO the challenge is to allow each statement to be in a different thread, and run concurrently and still get the same output for a and e.
  + Graphical user interface, application

    Description automatically generated
  + Text, application

    Description automatically generated with medium confidence
  + Notice above, how after we execute cin>>a, then a is available for other functions to use; and remember cin>>a was the first statement that needed execution in the combined program sequence given. We broadcast only because a only gets changed once with cin>>a; all other threads use a but do not update it.
  + Application

    Description automatically generated with medium confidence
  + Remember goal is concurrency and parallelism to maximize ability for those threads to run in any order (and in the environment of multiprocessing, they can run in parallel and the full function broken down into threads will thus be able to run faster than as 1 single process/thread.)
  + For this example, all variables are only updated once, so it makes this problem not as difficult; if a variable was updated more than once, then we would need to add in additional condition variables and pay attention to our use of a broadcast signal 🡪 may need to only signal for only one possible waiting thread to use a given value instead of all waiting threads.

# Question 7

* + Text, letter

    Description automatically generated
  + Text, letter

    Description automatically generated
  + Part b:
  + Text, letter

    Description automatically generated
  + If there is not enough balance, then the waiting thread will continue to wait, and all threads behind that thread will have to continue to wait in line until the FCFS thread at front of line is served.
  + Solution will be similar to this but not as complex:
  + Table

    Description automatically generated
  + We need to keep track of withdraws.
  + Deposit will never wait (has priority?); and remember monitor will already guarantee mutual exclusion; so, withdraw and deposit threads can only get into and use the monitor one at a time, however withdraw threads will have a condition variable so that it can wait in the event that there are not enough funds to withdraw.