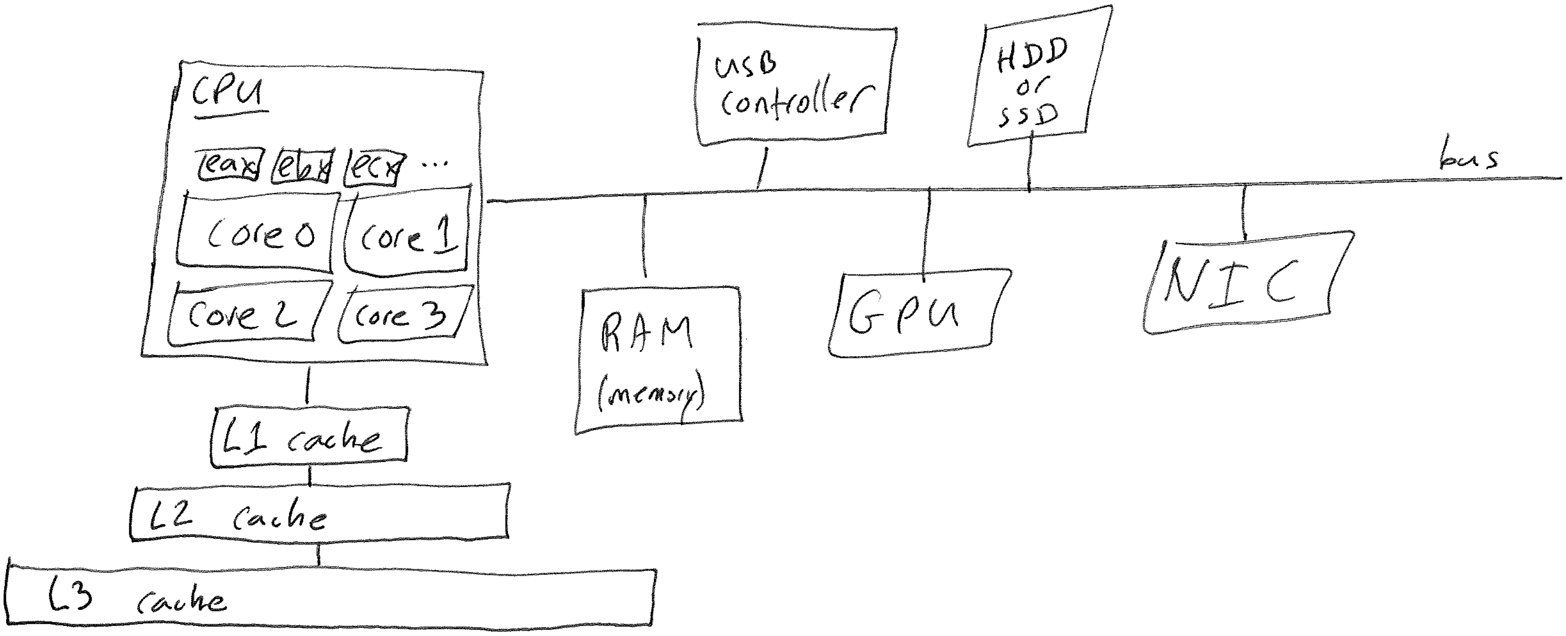
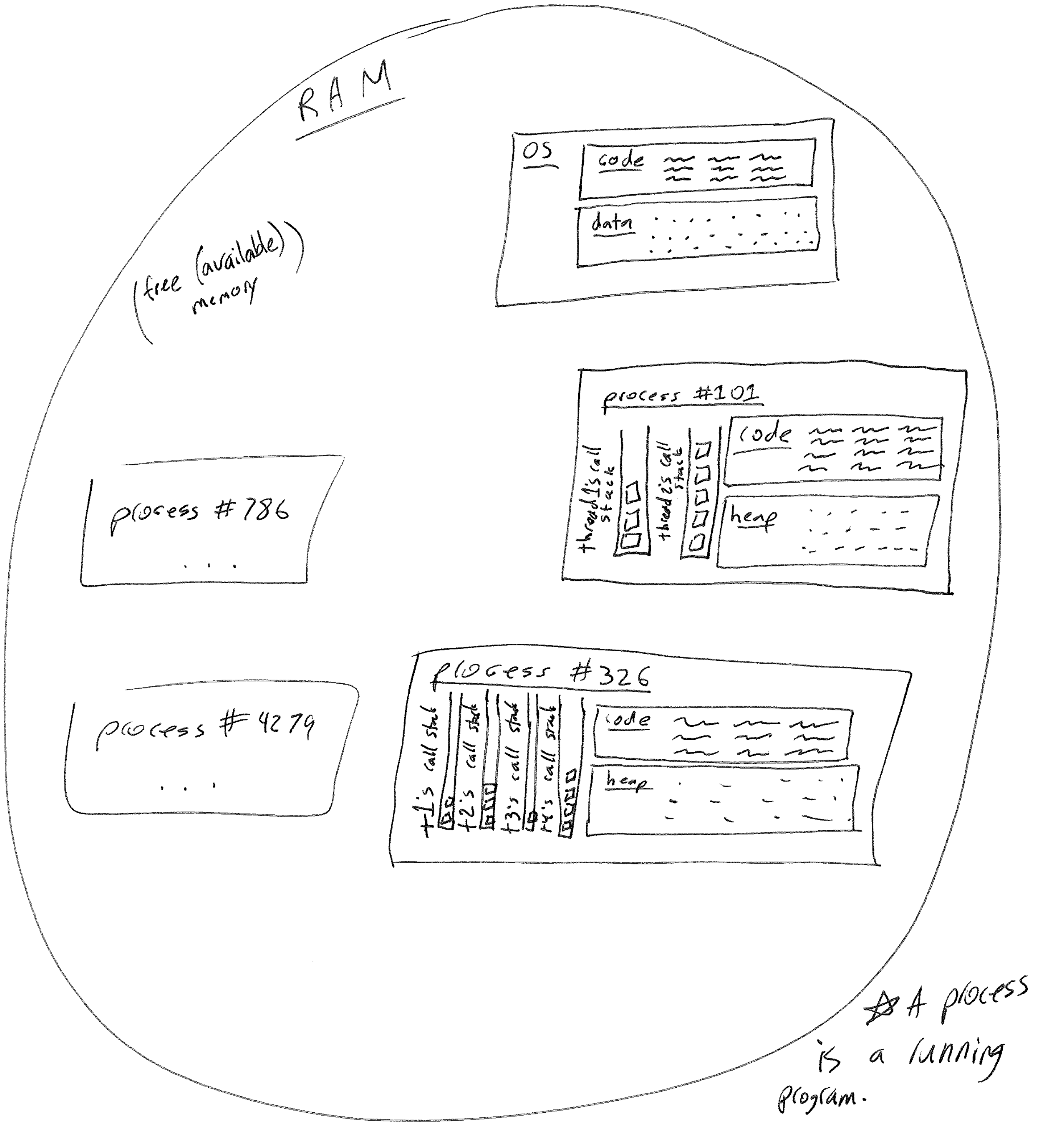
Parallel Programming

Let's draw a cartoon of our computer:



Notes:
- Usually each core has its own L1 and L2 cache.
- L3 is usually shared by all cores.
- On an Intel i7 CPU:
  - L1 : 32kB
  - L2 : 256kB
  - L3 : 8MB
- Every program is bottlenecked by one of these components!
  ↳ CPU-bound or IO-bound

Let's zoom in on the RAM and draw another cartoon.



Notes:

- The OS is just code and data just like any other program.
- One of the OS's jobs is to schedule threads onto CPU cores. The OS juggles all the threads and juggles itself in the mix!

Now we'll do some programming.

First, let's see how many cores, how much cache, and how much memory our Mac has.

Run 0.py → run do_stuff() from main thredd

→ check Activity Monitor

→ lauch two threads for do_stuff()

→ check A.M.

→ lanch 4 threads ...

→ check A.M.

→ Stop using numpy → just in put a while true: pass

→ check A.M → what happened? ... the GIL.

→ Swith to using processes.

→ check A.M.

→ Promise to show a threading gotcha and a multi-processing gocha.

Run 1.py → mention that printing the PID is often useful so that you know which process to kill if needed.

Run 2.py → Show how to pass arguments to threads, and how to join w/ threads.

→ Piont out that the ~~P~~PID is the same for all threads in a given process.

Run 3.py → Look at what 3_serial.py then 3_threads.py ~~is~~ doing... looks right?

→ Run repeatedly. What happens? → This is the threading gotcha!!

→ Let's switch to using processes. What happens? This is the multiprocessing gotcha!!!!!

↳ Explain fork().

→ Fix it to use a queue for results.

↳ live code it.

→ Compare runtimes of 3_*.py.

Final remarks:

- How many threads / processes should you launch?
  - A: the # of cores you have → look at your # core options on AWS.
- Why?
  - A: - because there's an overhead to context-switching between threads
      - ↳ the more threads, the more context switching is needed
    - Also, cache hits/misses. More misses w/ more threads.
- In Python, you'll probably choose processes over threads.
- Don't prematurely optomize. See 4.py for Donald Knuth's quote.
- This is only scratching the surface of parallel programming. Here are other technologies we won't touch, but that are widely used:
  - MPI: a very generic (flexible) distributed programming communication framework.
  - OpenMP: for parallelizing loops. (used in numpy very extensively)
  - GPU: a single GPU has thousands of cores!

Final Remarks (continued):

- Parallel programming is <u>hard</u>. Threads make it easy to currupt memory when you don't lock ~~p~~ critical areas properly. Processes have large overhead in communication - it's easy to actually <u>slow</u> down your program if you are communicating between processes inefficiently.

- There are times when it's easy to parallelize your code. In cases where your threads/processes don't need to communicate much, then it's very easy to parallelize. ~~These~~ Programs of this type are called "embarrassingly parallelizable". E.g. grid search.
  E.g. K-fold cross-validation

- Let's consider grid search... and let's consider using aws. Say one train-test iteration with your model takes 70 seconds. You want to search a hyper-parameter space of $5 \times 5 \times 5$ (grid search). But you want this to be finished in 20 minutes (you're leaving for lunch and will be back in 20 minutes and you want the search to be finished). How many cores do you need?
  ↳ $\boxed{\text{8 cores}}$ ← buy on ~~amazon~~ AWS. How much will that <u>cost</u> you?

Final Remarks (con't):

- Beware of fork(). It copies all the data in your ~~program~~ current process!

  Don't do this:

  1: Load all you data.

  2: Fork

  3: In each process, use part of ~~that~~ the data.

  Instead, do this:

  1: Fork

  2: Load only the data each process need.

  3: Use the data...

~~Appd~~ Appendix A:

	Threads	Processes
quick creation/ destruction	✓	
easy and fast access to shared mem.	✓	
robust to bugs/ errors		✓
not affected by Python's GIL		✓