Abstractions in Computer Systems

Interpreter:

1. The active element of a computer system.
2. Consists of three elements:
    a. Instruction reference: where to find the next instruction
    b. Repertoire: set of actions that the interpreter can perform
    c. Environment Reference: where to find the environment
3. Essentially, the abstraction is a Von Neumann architecture–it stores the instructions in memory along with the data
4. What variances do we see across these interpreters:
    a. What operation updates the instruction reference?
    b. What is the environment reference?
    c. What are the interrupts?
5. MetaPoint–Sometimes these abstractions are useful, even if they aren't 100% correct. Example: sometimes ADTs are best viewed by actions, sometimes best viewed by storage. The same might even be true generally.
6. Very often we have layers (e.g., Java program -> Java Interpreter -> Hardware)
7. Some programs will have multiple interpreters:

Communication Links:

1. Allows information to flow across physically separated modules
2. Interface:
    a. Send(name, message): send a message to name
    b. Recv(name, message): receive a message *from* name
        i. N.b., does not mean "force name to give me message", means "get a message if there is one"
    c. Sometimes, receive is really operation deliver(message), if you have an event-based program.
3. Send/Receive can have some different semantics to other things we've studied:
    a. In some cases, you can basically assume that send is reliable (e.g., memory bus).
    b. In other cases, you may never know if/when send was received (e.g., in a wide area network).
4. Why do we have both communication channels and memories?
    a. Could you implement a memory using a communication channel?
        i. YES! In fact, your computer does this with a memory bus.

Meta-Point–Sometimes these abstractions are useful even if they aren't 100% accurate. Last time we talked about how a DRAM subsystem can be viewed as a memory (duh!) but also as an interpreter. Here, we've seen that you can think about a DRAM subsystem as being a communication channel. It's all three! Depending on when you want it to look like something

Naming

1. Name–a symbolic way to refer to an object
2. Requirement for modularity and abstraction!
    a. Use in Interpreters – Instruction reference, env. Reference
    b. Use in Memory–"named object"
    c. Use in communication link–"named link"
3. Naming Scheme–A mapping from a namespace to a value-space:
    a. Name-space: alphabet of symbols and syntax rules
    b. Value-space: Universe of objects (objects could also be names!)
    c. Name-mapping algorithm: Associates names with values!
        i. Fundamentally must support value <- resolve(name)
        ii. Might support bind, unbind, enumerate, compare
4. Three main resolution strategies:
    a. Table Lookup
    b. Recursive: value-space might also include names!
    c. Multiple Lookup: Allow search through multiple contexts
5. Three fundamental problems in naming:
    a. Disambiguation–how do we handle duplicates? A: Contexts!
    b. Name vs. Value Lifetimes–How do we make sure that the name and value don't outlive each other?
    c. Fragility in names–how do we make sure that names don't over-constrict their values?
6. Three awesome use-cases:
    a. Late Binding (LD_Preload, Spark's Lazy execution)
    b. Data-Sharing (didn't get to in class)
    c. Unique ID Name Space (didn't get to in class)