We need metrics for gauging the quality of API in terms of modularity:

- performance (interfaces and modularity hurt performance)
- robustness (how well are errors and faults dealt with)
- simplicity (easy to use/learn/maintain)
- neutrality/flexibility/portability (API must make few assumptions to run on all systems)
- evolvability (needs to be able to improve; design for iteration)

Let's evaluate our interface for reading in the bare-metal application:

Success unclear $\downarrow$ Inflexible read size $\downarrow$

```c
void read_ide_sector(int sector, char *addr);
```

$\uparrow$ Must know sector size

Thus it is clear that this interface has many problems; Linux has the improved interface:

```c
char *readline(int fd);
```

This is still a bad design for an operating system in many ways; we can evaluate it according to our criteria to see

- Modularity:
  - assumes the system does memory allocation
- Performance:
  - unbounded work — can be very inefficient on long lines
  - un-batched — large overhead on short lines
- Robustness:
  - apps can crash on long lines
- Neutrality:
  - forces line ending convention
- Simplicity
  - quite simple

Thus it is clear that we need a more flexible and powerful API which will allow us to directly access low-level hardware. Linux includes the following:

```c
// lseek shifts the read and write point, white read loads the data
off_t lseek(int fd, off_t offset, int flag);
ssize_t read(int fd, char*addr, size_t bufsize);

// these have now been implemented as one primitive:
char *pread(int fd, char *buf, size_t bufsize);
```

Ultimately, we can see that the main difference between the two API's comes down to memory allocation — choice of API is incredibly important! Our procedure call API is not the only place where modularity is important, however; suppose we have the following source code:

```c
int factorial(int n) { return (n == 0 ? 1 : n * factorial(n-1)); }
```

and we run the following commands:

```
$ gcc -S -O1 fact.c
$ cat fact.s
```

which results in the following output assembly code:

```
fact:
  movl $1, %eax
  testl %edi, %edi
  ne .L8
  ret

.L8:
  pushq %rbx
  movl %edi, %ebx
  leal -1(%edi), %edi
  call fact
  imult %ebx, %eax
  popq %rbx
  ret
```

What could go wrong? Consider a malignant actor runs the following assembly code:

```
badfact:
  movq $0x39c54e, (%rsp)
  ret
```

We could prevent this by implementing *Hard Modularity*

- no trust is necessary (nor often exists)
- one (or both) modules are protected
- avoids fate

as opposed to the currently used *soft modularity*

- the caller and callee are part of "one big happy family"
- a cheap but unsafe approach
- operates according to the caller/callee contract:

Contract	**Violation Consequence**
Callee only modifies its own variables and variables it shares with the caller; it does not modify the stack pointer and leaves the stack the same as it was called.	Callee corrupts the caller's stack and the caller may use incorrect values in later computations.
Callee returns to the caller.	Callee gets unexpected values of loses control of the program.
Return values are stored in %eax.	Callee will receive whichever value is in %eax, which may be incorrect.
Caller saves values in temporary registers to stack before calling callee.	Callee may change values that caller needs and caller will receive an incorrect result.
Callee will not have a disaster that affects caller (ex: early termination).	*Fate Sharing*: caller will also terminate.

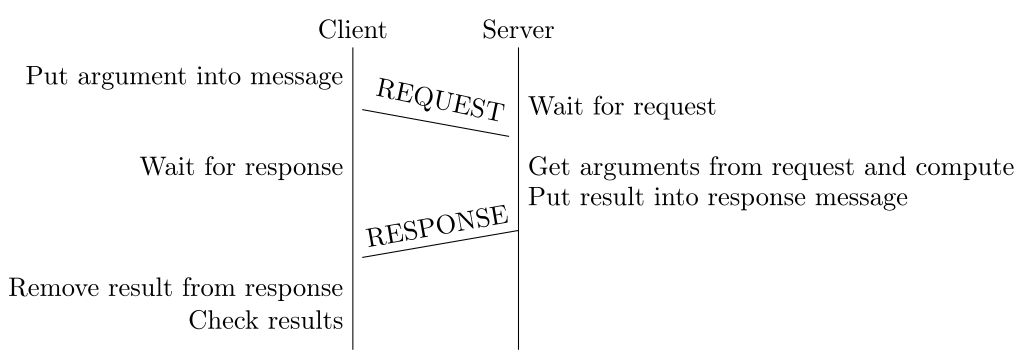
We can apply modularity to all 3 of the fundamental abstractions:

1. Interpreters/function calls (soft)
   - run untrusted module on a software "machine"
   - have 3 components:
     (a) instruction reference: location of next instruction
     (b) repertoire: set of actions and instructions

    (c) environmental reference: tells where to find environment

2. Processors:
   - a general purpose implementation of an interpreter
   - Instruction reference is a program counter
   - Repertoire includes ADD, SUB, CMP, and JMP; LOAD not READ
   - Have a stack and wired environmental reference
3. Java Virtual Machine (utilized by SEASNET)
   - Unprotected:
     (a) communicates via shared memory → SLOW
     (b) run out of stack space
     (c) functions can step on each other's memory
     (d) infinite loops
4. Virtualization (hard)
   - we simulate the interface of a physical object by:
     (a) creating many virtual objects by multiplexing one physical one
     (b) creating one virtual objects by aggregating multiple physical ones
     (c) implementing a virtual object from a physical one by emulation
   - some hardware and assembly is involved
   - limited to a preset service
5. Client/Server (hard)
   - Details:
     - run each module on its own machine
     - communicate via network messages
     - no reliance on shared state; global data is safe
   - Transaction is arms' length, so errors do not propagate
     - error propagation is called fate sharing
     - since there is none, all of our errors are controlled
   - Client can place limit on services
   - Encourages explicit, well defined interfaces

We can visualize the client/server interaction as follows:



We focus on number 2, and number 3 is covered in CS 118.