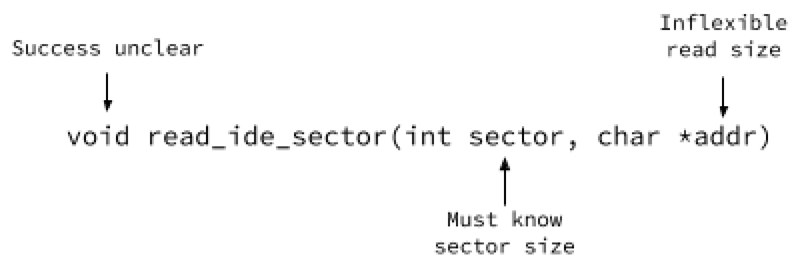
3. Modularity

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

- performance (interfaces & modularity hurt performance)
- robustness (how well are errors & 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 look at our interface for reading in the bare-metal application:



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

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

- This is a BAD DESIGN for an OS
  - Modularity
    - assumes the system does memory allocation
  - Performance:
    - unbounded work - may take forever for big lines
    - un-batched work - overhead for small lines
  - Robustness:
    - apps crash with big lines
  - Neutrality:
    - forces line ending convention
  - Simplicity:
    - good simplicity

—> we a more flexible and powerful API which can allow for direct access:

```
// lseek moves the read and write point, and read reads data!
off_t lseek(int fd, off_t offset, int flag);
ssize_t read(int fd, char *addr, size_t bufsize);
// this has now been implemented directly ->
char *pread(int fd, char *buf, size_t bufsize);
```

So why is read() is good and readline() is bad?
~ because of memory allocation—choice of API matters!

But our procedure call API is not the only place where modularity is important
Source code:

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


$ gcc -S -01 fact.c
$ cat fact.s
```

Assembly:

```
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
```

So what could go wrong with our factorial function?

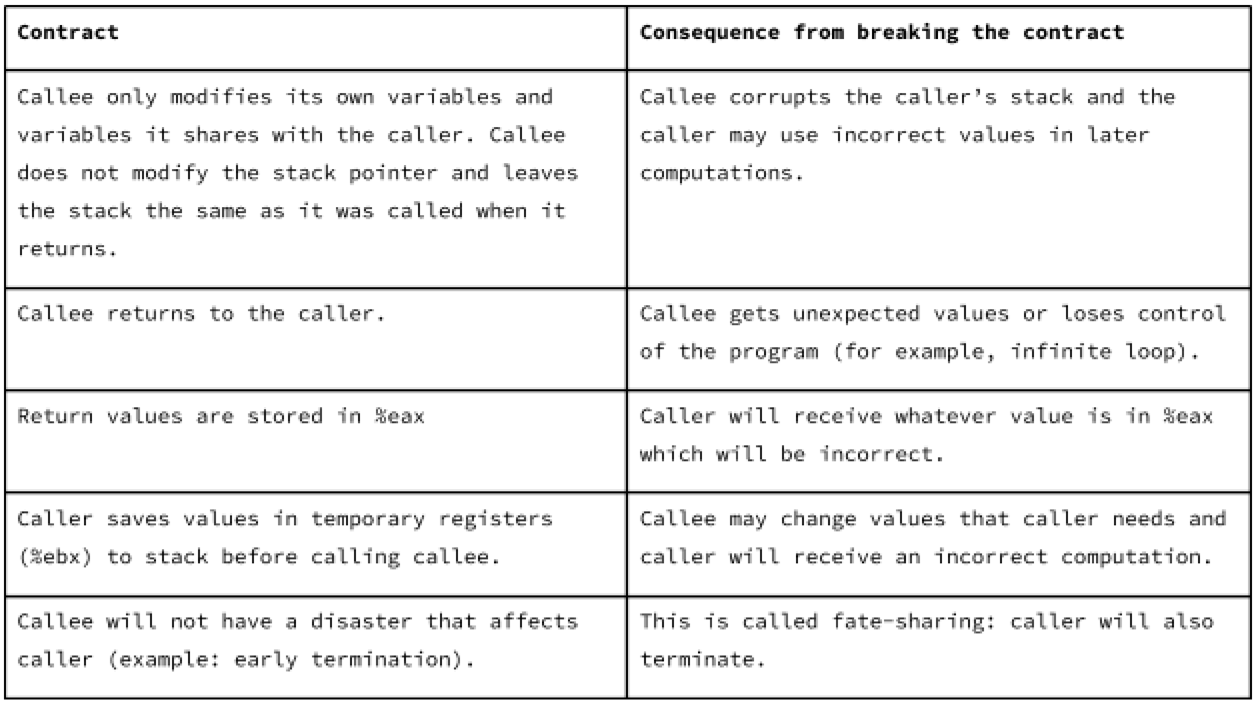
```
badfact:
  movq $0x39c54e, (%rsp)
  ret
```

& we jump to the wrong return address!

This is because function calls utilize:

Soft Modularity

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



Contract	Consequence from breaking the contract
Callee only modifies its own variables and variables it shares with the caller. Callee does not modify the stack pointer and leaves the stack the same as it was called when it returns.	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 or loses control of the program (for example, infinite loop).
Return values are stored in %eax	Caller will receive whatever value is in %eax which will be incorrect.
Caller saves values in temporary registers (%ebx) to stack before calling callee.	Callee may change values that caller needs and caller will receive an incorrect computation.
Callee will not have a disaster that affects caller (example: early termination).	This is called fate-sharing: caller will also terminate.

the caller/callee contract

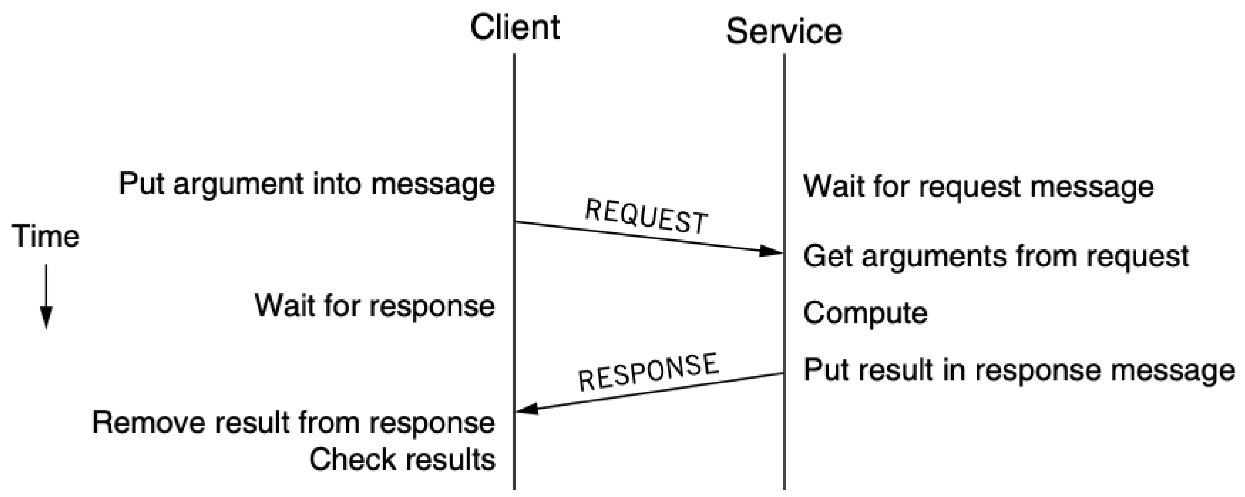
Hard Modularity

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

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

1. **Interpreters/function calls** (soft)

   a. run untrusted module on a software “machine”
   b. have 3 components:
      i. instruction reference: location of next instruction
      ii. repertoire: set of actions & instructions
      iii. environmental reference: tells where to find environment
   c. Processors:
      i. a general purpose implementation of an interpreter
      ii. Instruction reference is a **<u>program counter</u>**
      iii. Repertoire includes ADD, SUB, CMP, & JMP; LOAD not READ
      iv. Have a stack & wired environmental reference
   d. Java Virtual Machine (utilized by SEASNET)
   e. Unprotected:
      i. communicates via shared memory → SLOW
      ii. run out of stack space
      iii. functions can step on each other’s memory
      iv. infinite loops
2. **Virtualization** (hard)
   a. we simulate the interface of a physical object by:
      i. creating many virtual objects by **<u>multiplexing</u>** one physical one
      ii. creating one virtual objects by **<u>aggregating</u>** multiple physical ones
      iii. implementing a virtual object from a physical one by **<u>emulation</u>**
   b. some hardware & assembly is involved
   c. limited to a preset service
3. **Client/Server** (hard)
   a. Details:
      i. run each module on its own machine
      ii. communicate via network messages
      iii. Client & service do not rely on shared state, so global data is safe
   b. Transaction is arms’ length, so errors do not propagate
      i. error propagation is called fate sharing
      ii. since there is none, all of our errors are controlled
   c. Client can place limit on services
   d. Encourages explicit, well defined interfaces


Client
Service
Time
Put argument into message
REQUEST
Wait for request message
Get arguments from request
Wait for response
Compute
RESPONSE
Put result in response message
Remove result from response
Check results

We focus next on #2, (#3 is covered in CS 118)