* syscall cont.

executing syscalls (b/c it's in a different addr space), is similar to
remote procedure calls (RPCs).

```
//////////////////////////////////////////////////////////////////////
/// verifying args

// example syscall: read(2)
int sys_read(int fd, __user void *buf, int size)
{
  // 1. verify args
  // fd:
  // - valid range? from 0 to some "MAX_FD"
  // - fd might not even be an opened file
  // - was file opened for reading?
  // - (does user process have permissions to read file? often checked at
  //    open time)

  // size: check not negative
  // - some OSs may have a max size of allowed read unit

  // buf (void* pointer -- belongs to user, virtual address!):
  // - not NULL
  // - is it allocated? -> is it in a valid addr space
  // kernel manages memory in 4KB (or PAGE_SIZE) units.  Each page has
  // flags: Read, Write, eXecute.  Protections are on the Virtual page addr.
  // Kernel needs to verify it can write to user addr.
  // Kernel translates the start addr of buf into the virt. page it belongs
  // to, then checks the virt-to-phys map for THIS process, to check if
  // there's even a mapping (else return error -EFAULT).
  // If there's a mapping, then verify access permissions to the page, else
  // get error (EFAULT, EPERM).
  // Kernel has to take buf, extend it to its enclosing 4K pages, then check
  // EACH of those pages for existence and permission.
  verify_area(buf, len, VERIFY_READ|VERIFY_WRITE);

  // 2. do actual work
  // - finds data from disk and file system, get I/O into memory
  // - kernel writes data from I/O buffer into user buffer,
  // - kernel cannot write directly to "__user void* buf"!  Must write into
  // that buffer's physical location.

  // 3. cleanup

}

int sys_open(__user char *file, ...)
{
  char *kstr;
  // verify string: POSIX says PATHNAME_MAX is 4096
  // at most need to verify 2 pages.
  // then can translate virt to phys, read bytes to find null terminating
  // the string.
  // Best to copy string into kernel's own buffer. Like strdup(3).
  kstr = getname(__user char *str); // performs kmalloc, need to kfree
  // do putname when done with kstr.

  // can do it yourself using verify_area, then kmalloc, then copy from user
  // to kernel using:
  copy_from_user(void *kaddr, __user void *uaddr, int len);
  // or
  copy_to_user(__user void*u addr, void *kaddr, int len);
}
```

```
//////////////////////////////////////////////////////////////////////
/// CODING STYLE

// ex. syscall needs to open 2 files to read, open 3rd file to write, and
// malloc a buffer.
// too much nesting, hard to code, fragile, bug prone, duplicates too much code
int sys_foo1(char *f1, char *f2, char *f3)
{
  void *kbuf;
  struct file *fp1, *fp2, *fp3;

  // 1. verify args

  // 2. prepare and initialize
  fp1 = filp_open(f1, O_READ, ...);
  if (failed to open fp1) {
    return -ERRNO; // return appropriate error
  }
  fp2 = filp_open(f2, O_READ, ...);
  if (failed to open fp2) {
    flip_close(fp2);
    return -ERRNO; // return appropriate error
  }
  fp3 = filp_open(f3, O_WRITE, ...);
  if (failed to open fp3) {
    flip_close(fp1);
    flip_close(fp2);
    return -ERRNO; // return appropriate error
  }
  kbuf = kmalloc(4096);
  if (kbuf == NULL) {
    flip_close(fp1);
    flip_close(fp2);
    flip_close(fp3);
    return -ENOMEM;
  }

  // main body of code

  // cleanup

}

// v2: w/o too much nesting
// benefits: single point of exit, no deep nesting, no duplicated code
// easier to code, less bug prone
int sys_foo2(char *f1, char *f2, char *f3)
{
  void *kbuf;
  struct file *fp1, *fp2, *fp3;
  int retval; // maybe initialize to something if needed

  // 1. verify args

  // 2. prepare and initialize
  fp1 = filp_open(f1, O_READ, ...);
  if (failed to open fp1) {
    retval = -EPERM; // or appropriate error
    goto out;
  }
  fp2 = filp_open(f2, O_READ, ...);
  if (failed to open fp2) {
    retval = -ERRNO; // return appropriate error
    goto out_close1;
  }
  fp3 = filp_open(f3, O_WRITE, ...);
  if (failed to open fp3) {
    retval = -ERRNO; // return appropriate error
    goto out_close2;
  }
  kbuf = kmalloc(4096);
  if (kbuf == NULL) {
    retval = -ENOMEM;
    goto out_close3;
  }

  // main body of code
  // do what needs to be done, then if succeeded, fall through to next label
  // (out_kfree).  Just remember to set retval = 0 (or whatever notes success)
  // if there's a failure inside MIDDLE of main body of code: goto
  // out_kfree.

  // cleanup
 out_kfree:
  kfree(kbuf);
 out_close3:
  flip_close(fp3);
 out_close2:
  flip_close(fp2);
 out_close1:
  flip_close(fp1);
 out:
  return retval;
}
```