30 Mini-Sermon #1: Simplicity

There are few absolutes in Computer Science. When building computer systems, most things involve tradeoffs. One exception is simplicity.

Forces against simplicity:

- Intelligence = ability to do things others can't understand? Better algorithm must be more complicated?
- Advertising: make your product have more features than anybody else's product?

Reasons for simplicity:

- Sales: can't sell to a confused customer.(?!?)
- Easier to build, maintain. Ability to make systems more powerful is limited by complexity, both for designers and users. By finding simple ways of doing things, can add more power before system becomes unmanageable. UNIX started out incredibly simple, which made it easier for the system to grow later.
- Faster: e.g. RISCs.
- Cheaper.

How to get simplicity:

- Don't accept complexity!
- Ask "what is the simplest it can possibly be?" and "suppose we throw out this feature: what do we lose?" Don't keep anything around unless it's truly important. When in doubt, leave it out.
- Study different systems to see how they are alike and how they are different. Try to find a single, simple way of doing all the common things.

Simplicity and abstraction are really two words for the same thing.

Instead of using creativity to confuse people, use it to find easy, "obvious" ways of doing things. Ultimate compliment is when somebody tells you "Heck, I could have thought that up!"

31 Mini-Sermon #2: Building High-Performance Systems

Dilemma:

- In operating systems, performance is usually a critical issue.
- Usually, only a few pieces of OS are performance-critical (e.g. context switch time).
- Doing performance optimizations makes code messier, harder to understand and maintain. Don't want to do it unless absolutely necessary.

Option #1: Measurement. Grab all data that can be gotten from existing systems. Measure process sizes, file distributions, etc etc. Try to identify areas where performance does and doesn't matter.

What about things with no existing data?

Option #2: Modelling. Build model of system, write simulator to simulate system behavior in order to see how well your algorithms will work. Problem: it's often hard to figure out right system model: extrapolate from current systems.

Option #3: Tuning.

- Make system as simple as possible, but build in mechanisms for evaluating performance once system is running.
- Get system running, then make performance measurements.
- Tune those portions of the system that didn't perform up to snuff. This means that strategy algorithms must be designed to be changeable.

Above all, don't worry about performance unless you're SURE there's a problem. In most places it doesn't matter.

Remember: Every feature or change should be justified. Don't do things just because they're "cute" or "neat".

32 Mini-Sermon #3: Information as Property?

Lots of recent attention given to computer break-ins

Typical reactions:

- It's funny. Those kids sure did make those adults look dumb.
- It's just part of growing up. Leave 'em alone, they'll grow out of it (every kid should be allowed to transfer 3 billion dollars out of B of A on his/her way through puberty).
- It's a sign of brilliance.
- They didn't hurt anything, they were just looking around.
- People didn't protect their accounts properly, so they deserved what they got.

Yale cheating case was thrown out because only copied, didnt' steal, and because prof "didn't adequately protect" stuff.

Why is information different from other property?

- Is it OK for someone to enter your house if you leave a window open, and rummage through all your personal belongings, as long as they put everything back where they found it?
- Is it OK for someone to hot-wire your car and take it for a joy-ride, as long as they fill the gas tank when they return it?
- Is it OK to break into a bank vault and examine contents of all safety-deposit boxes, as long as nothing is stolen?
- If that's OK, then what if only .01% is stolen? Is that OK too? Where is the line to be drawn?

Information is property, and must be protected as such. Need to do three things:

- Make systems more secure.
- Change societal attitudes. This isn't funny anymore.
- Establish laws to protect information in a reasonable way. ?!?

33 Mini-Sermon #4: Programming and Documentation Style (Discipline and Craftsmanship)

Question is: how to build large software systems in a reasonable amount of time, and make them work reliably?

Books and courses on SE tend to discuss life cycles, PERT charts, staffing, and other technical approaches. I'm going to talk more about philosophical details.

Programming is a craft: something like a high-tech form of silversmithing. To make big systems work, it takes a tremendous amount of discipline and structure. If you're good at it, you'll have a tremendous advantage.

Rule #0: simplicity. It's easy to make things complicated, harder to make them simple. Don't accept complexity.

Rule #1: adopt a consistent style and use it everywhere. Decide on file organization, procedure structuring, naming conventions, location of curly braces, everything. By doing things consistently, they don't get omitted.

Rule #2: don't litter. Temptation is to make fast fixes that dirty things up. It's crucial to take the time to fix things right at the beginning. You'll never have time to come back to it later!

Rule #3: document carefully as you go. Don't put this off! Most important things are interfaces: procedure headers and data structures and other things that tie together the parts of the system. Tell about writing procedure headers for a whole module before writing the procedures.

Rule #4: documentation should describe things at a higher level than code (but not *too* high a level). Examples of pathological cases:

- "Add one to i."
- "Now we have it the way we want it."

Rule #5: Put documentation near the code: otherwise you forget to change the documentation when you change the code.

Rule #6: Quality, not quantity. Don't need enormous amounts of documentation if what you have is high-quality. E.g. don't need 400-character variable names.

Summary: discipline, craft. Take time up front to save time later.