THE

CS380L: Mike Dahlin

September 4, 2008

Be aware of the fact that experience does by no means automatically lead to wisdom and understanding; in other words, make a conscious effort to learn as much as possible from your previous experiences
– E. W. Dijkstra. "The Structure of the 'THE' Multiprogramming System." *Communications of the ACM*, 11(5), May 1968, pp. 341-346

1 Preliminaries

1.1 Review

- Multics: key ideas
  - Specific technologies: fine-grained sharing, dynamic libraries
  - Principles
    - Designing for security: time tested principles (failsafe defaults, complete mediation, ...)
      - Challenge: Psychological acceptability...KISS
    - Overall design:
      - Unified naming and addressing
      - Fine grained sharing
      - Dynamic linking
      - Autonomy ("Modularity")

1.2 Outline

- THE: System structure
- Admin: Project
- THE: Perls of wisdom

1.3 Preview

- Historical perspective: UNIX
- Minimal abstractions: exokernel, scheduler activations
- Virtual machines: Disco, ESX

2 Multix v. THE v. Unix

Purpose of these papers in class
All provide useful historic context – how architecture of modern OS coalesced.
"Flavor" of papers very different.

- Multix
  - Important (and difficult) technical content
  - Research approach: Ambition. Build the ideal system. Push idea (sharing) to extreme
- THE and Unix
  - Simplicity. Humility. Build something that works.
  - Research approach
    - THE: Scientific approach to design
      - Also...
      - Push *how to build correct system* to an extreme
      - → new abstractions (semaphore)
    - Unix: "Good taste" in design
      - "What are the right abstractions'?"
      - Lampson: Hints for computer system design
        The designer usually finds himself floundering in a sea of possibilities, unclear about how one choice will limit his freedom to make other choices, or affect the size and performance of the entire system.
      - How to teach this?
      - Apologies – lots of philosophical musings today compared to last time (and most of future meetings.)
      - Probably because I don't know how to teach this...
- Easy to see the contributions of mutix – plenty of "hard stuff." Next two papers – THE and Unix – have a different flavor. Very little seems "hard." Much seems "obvious in retrospect." Many of the lessons are broader than "use this mechanism to solve this problem." Instead, many of the lessons are about how to build systems that actually work – "taste", "discipline," "simplicity." The success of these systems came from "fear" and from the ability to choose just the right subset/variation of ideas that had been floating around in the community.

  Two reasonable approaches to research – push an idea to the extreme, build something that works. Try to do both?
- I feel I am very good at the multix style of research and pretty good at the THE style. I wish I was better at the THE and Unix style.
- I *have* gotten *much* better at the "mechanical" process of building systems that work ("only trivial 10-minute to find and fix bugs"). Mostly from learning the hard way (doing it wrong and learning "fear"). Some from believing advice like that in this paper. The latter is easier.
  - I certainly have gotten better at "given a spec, spend enough time on design so that building/debugging easy"

- Maybe this also enforces a discipline of creating a spec that is sufficiently "humble"
- (So maybe one way to teach/learn the hard idea of "good taste" is to practice some of (I think) straightforward pieces of advice on how to build working systems in this and other papers.)

- QUESTION: Do you think successful class projects likely to look more like Multix or THE/Unix?
  - *Hint* Dijkstra: "having very limited resources (viz. a group of six people of, on the average, half-time availability) and wishing to contribute to th art of system design...we were faced with the problem of how to get the necessary experience."

3 THE

3.1 Historical perspective

- Written at a time when OS complexity running amuck
  - IBM System 360 announced 1964, OS/360 delivered 1967;
  - IBM System 370 introduces virtual memory (1972)

- Influential paper:
  - Overall architecture and approach: proposes "scientific" approach to OS design and implementation
  - Specific abstractions:
    * Cooperating sequential processes
    * Synchronization
    * Virtual memory

3.2 System structure

- Core contribution: strict hierarchy of sequential processes → can reason about correctness of system

1. Strict hierarchical structure
2. Sequential processes – "succession of the various states has a logical meaning but not the actual speed with which the sequential process is performed."
   - undefined speed ratios
   - **semaphores** – explicit mutual synchronization statements
   - **Hugely influential** – a formal way to reason about concurrency.
     * Obvious (today). At the time, other systems assumed time bounds from, say, an interrupt occurred until when it was serviced. Problems: (1) complexity ripples through the system (now need to "protect" these assumptions throughout your design – not modular, (2) difficult to test. I seem to recall Dijkstra writing about a contemporary IBM OS that occasionally would lose a character of input because it was written in this style, but I cannot find the cite.

- Another key contribution: clean definition (and one of the first working implementations) of virtual memory abstraction
  - Virtual addressing – separation of logical address ("segment" *Note: not same as current segment – fixed size virtual page*) and physical address ("page").
- The hierarchy

level 0 processor allocation (*sequential processes*) – "above level 0 the number of processors actually shared is no longer relevant."

level 1 Segment controller (*virtual memory*) – "at all higher levels identification of information takes place in terms of segments, the actual storage pages [have] lost their identity [and have] disappeared from the picture."

level 2 message interpreter – demultiplex consoles to processes – "Above level 2 it is as if each process had its private conversational console."

level 3 buffering of streams

level 4 user-level programs

level 5 the operator

3.3 Evaluation of "strict" hierarchical structure and simplifying assumptions

- If one can enforce strict hierarchy reasoning about system becomes simpler
  - The above statement is not controversial (right?). The question is: can one enforce a strict hierarchy?
- Can sophisticated systems be constructed as strict hierarchy (or is THE a toy?)
  - *Industrial software makers tend to react to the [THE] system with mixed feelings. On the one hand, they are inclined to think that we have done a kind of model job; on the other hand, they express doubts whether the techniques used are applicable outside the sheltered atmosphere of a University and express the opinion that we were successful only because of the modst scope of the project.* – EWD
  - *I should like to venture the opinion that the larger the project, the more essential the structuring!* – EWD
  - In practice, few subsequent systems are *strict* hierarchies.
  - Two potential hypotheses:

  1 Hypothesis: THE is a toy that gets much of its simplicity b/c of limited scope
  What is missing from THE that simplifies?
    * Sharing (no shared files, let alone Multix style fine grained sharing.)
      · Does sharing preclude strict hierarchy?
    * What else? (file system, networking)
    * Which would force change to more complex structure? Can you characterize what class of features admits/prevents strict hierarchy? Can such features be recast/limited to allow hierarchy?

· Networking often considered classical example of when callbacks useful...

2 Hypothesis: Much of the complexity of subsequent systems comes from laziness/poor design. If their designers were as smart/"scared" as EWD, these systems also could be simplified.

· Requires willingness to cut features as well as restructure internals?

· See "intellectual level" quote below.

– Challenge: choosing features:

* EWD was willing to give up some really enticing features (e.g., sharing), which allowed them to focus on solving an important problem well

· *really* difficult to give up features (sharing, what else?). *really* important to do so on research project

· counterexample: fine-grained consistency in PRACTI

* On other hand, if you give up too many features, maybe you just have a toy?

· What if they give up VM, still impressive?

· What if they give up multiple tape readers and punches, still impressive?

· Give up teleprinters, plotter, line printer? Still impressive?

– What lessons to apply to systems we build?

Mike's take: *start with hierarchy. Regard decision to break hierarchy as* **grave decision**. *Whole team should debate such a decision vigorously. Alternatives, including omission of features, should be explored. Can you "prove" that a system that implements feature X requires non-hierarchy? (Trying to do so may help you put your finger on bad assumption/decision? Speculate: at least 90% of time, hierarchy could be retained?)*

Since I don't fully understand when non-hierarchy required, conscious decision (1) avoid if not needed, (2) allows later reflection (and perhaps wisdom) on decision (to better balance pros/cons in future and/or to learn what the general rule is.)

- Think about non-hierarchical structures in systems we discuss in class. What, if anything, was gained?

3.4 Project management

- Dijkstra describes 3 phases of project: Design, code, test

– For a typical research project, what ratios in planned time spent?

```
<-design-><------ code -------><t>
```

* Design: not including designing tests
* Code: not including writing test code
* Test: little or no time planned for systematic testing; will find bugs as they come up
* Code time dominates plan

· E.g., class project "1 month to figure out what to do, 1 week design, 4 weeks code (but we know it will end up taking 6-8), 1 week to run experiments"

– For a typical research project, what ratios in actual time spent?

```
<dsgn><---------- code ----------><---- debug/recode/redesign -------...>
```

  - An hour or two of high level design (little gets written down)
  - Coding takes much longer than expected (and includes lots of redesign)
  - Debugging takes *much* longer than expected (and includes lots of re-implement/redesign)

- For THE, what was ratio of time spent/intellectual effort?

```
<------design--------><code><--- test ---->
```

  - Lots of time spent on design
    - Refuse to accept complexity
    - Develop ways to reason about the system
    - **Design with testing in mind**
    - *The designer's responsibility to construct his mechanisms in such a way—i.e. so effectively structured—that at each stage of testing procedure the number of relevant test cases will be so small that he can try them all and that what is being tested will be so perspicuous that he will not have overlooked any situation.*
    - IMHO: Current commercial practice often splits testing from programming ("don't waste valuable programmers' time testing...have low-level testings staff for that.") Consequence: Programmers are insulated from the negative consequences of their bad design/implementation decisions. Contrast that with THE focus on designing an architecture that can actually be tested...
  - Coding "routine"
  - Test: "A major intellectual effort"
    - The reward: all bugs are simple of the simple "10-minute" variety
    - You hear the quote:
      "The only errors that showed up during testing were trivial coding errors...each of them located within 10 minutes (classical) inspection by the machine and each of the correspondingly easy to remedy"
    - Your next question should be "What price do I have to pay/who do I have to kill to live in that world?"

- My experience (soapbox)
  - For a long time: typical pattern
  - Last few years: only get to write code around critical paper deadlines – e.g., I've got 2 weeks to get an experiment or subsystem written.

  → Only have time to do it if it works the first time

  → Don't bother to start coding unless I know I will be able to finish in time to matter (and before other responsibilities swamp me again)

  - Typically now about 50/30/20 design/code/test
  - It's been a long time since I saw a "major" bug in my code
    - Why: design for simplicity – when I design I worry "is this simple enough that I am certain there will be no bugs or unexpected interactions"? If not, redesign.
      - Modularity is king – *design* your system and your interfaces so that you can reason about each module independently

    - * Rule of thumb: 1 hour of design time saves 10 of implementation/debugging. (motion picture storyboard analogy)
  - Don't believe me?
    - * I won't ask you to change what you do for the rest of your life on faith
    - * I will ask you to invest 10-20 design hours before starting to write code just this once
    - * If I'm wrong, you've wasted 20 hours
    - * If I'm right, the 20 hours spent now will double your experimental research productivity
    - * This seems like a good bet!

- This works for papers too – design (outline) then build (write)
  - Use outline to detect and eliminate "complexity" and "skipped steps" in argument
  - Design review: Talk through the outline with other members of team (use LCD projector); then with someone outside of the team.

4 Admin

4.1 Sermon: Referee

Note: right column of schedule includes "professional development" papers

Read Smith "Hints for Referee" – how to evaluate/critique papers (whether you are a refereee or want to learn from paper.)

- *If you want to be taken seriously as a referee, you must have a middle-of-the-road view–you must be able to distinguish good from bad work, major from minor research, and positive from negative contributions to the literature. A referee who always says "yes" or always says "no" is not helpful.* A.J. Smith

- Report structure
  - "briefly state your recommendation and the reasons for it"
  - "summarize the point of the paper in one to five sentences, both for the editor's use and to ensure that you actually understand the paper"
  - "you should evaluate the validity and significance of the research goal"
  - "evaluate the quality of the work (methodology, techniques, accuracy, and presentation)"
  - "provide an overall recommendation for or against publication"
  - Recommended changes
    - * Usually more detailed for good papers than bad ones

- Categories of paper (quoting Smith)

1. Major results; very significant (fewer than 1 percent of all papers).

2. Good, solid, interesting work; a definite contribution (fewer than 10 percent).
3. Minor, but positive, contribution to knowledge (perhaps 10-30 percent).
4. Elegant and technically correct but useless. This category includes sophisticated analyses of flying pigs.
5. Neither elegant nor useful, but not actually wrong.
6. Wrong and misleading.
7. So badly written that technical evaluation is impossible.

- Other thoughts (Mike)
- Structure of review (TBD: Need to merge this structure with smiths. Both have good points..)
  - Summarize what they did; claimed contribution
  - 3 arguments for
  - 3 arguments against
  - recommendation
  - detailed comments
- Bar is lower for new directions than refinements on old ideas

4.2 Hamming: You and your research

- Goal: Do *great* research

  > As most people realize, the average published paper is read by the author, the referee, and perhaps one other person. Classic papers are read by thousands.
  >
  > Most scientists spend almost all of their time working on problems that even they admit are neither great or are likely to lead to great work

- Goal is: work on and then solve the important problems
  - Daunting? Only work on "nobel prize" problems?
  - No: Need a good attack, "some reasonable idea of how to begin."
- Strategy 1: Know the important problems

  > Great scientists all spend a lot of time and effort in examining the important problems in their field. Many have a list of 10 to 20 problems that might be important if they had a decent attack. As a result, when they notice something new that they had not known but seems to be relevant, then they are prepared to turn to the corresponding problem, work on it, and get there first.

- Strategy 2: Cultivate deep understanding of many things

  > Knowledge and ability are much like compound interest – the more you do the more you can do, and the more the opportunities are open to you.

The above story also illustrates what I call the "extra mile." I did more than the minimum, I looked deeper into the nature of the problem. This constant effort to understand more than the surface feature of a situation obviously prepares you to see new and slightly different applications of your knowledge.

It took me a long time to realize that each time I learned something I should put my "hooks" on it.

The evidence is overwhelming that steps that transform a field often come from outsiders.

Luck favors the prepared mind. – Pasteur

- Strategy 3: Work with passion

  Hard work is a trait that most great scientists have. Edison said that genius was 99% perspiration and 1% inspiration. Newton said that if others would work as hard as he did then they would get similar results.

  When I first met Feynmann at Los Almos during WWII, I believed that he would get a Nobel prize. His energy, his style, his abilities, all indicated that he was a person who would do many things, and probably at least one would be important.

- Strategy 4: Sell your ideas

  Too many scientists think that

  $$sellingtheirideas$$

  is beneath them, that the world is waiting for their great results. In truth, other researchers are busy with their own work. You must present your own results so that they will stop their own work and listen to you. Presentation comes in three forms: published papers, prepared talks, and impromptu situations. You must master all three forms.

  Lots of good work has been lost because of poor presentation only to be rediscovered later by others. There is a real danger that you will not get credit for what you have done. I know of all too many times when the discoverer could not be bothered to present things clearly, and hence his or her work was of no importance to society.

4.3 Project

5 Perls of wisdom

- Most of the introduction is "lessons"
  - *Be aware of the fact that experience does by no means automatically lead to wisdom and understanding; in other words, make a conscious effort to learn as much as possible from your previous experiences.*

  - *The intellectual level needed for system design is in general grossly underestimated. I am convinced more than ever that this type of work is very difficult, and that every effort to do it with other than the best people is doomed to either failure or moderate success at enormous expense.*
  - *One remark is that production speed is severely slowed down if one works with half time people who have other obligations as well. This is at least a factor of four; probably it is worse. The people themselves lose time and energy in switching over; the group as a whole loses decision speed as discussions, when needed, have often to be postponed until all people concerned are available.*
    * **Structure your life to be productive.**
    * Tactic: Structure your day to have uninterrupted block of time. Guard it jealously. Read e-mail at 1 to 3 scheduled times during the day. Avoid the web (or schedule 1 to 3 breaks).
    * Tactic: Sit near your collaborators. (Don't try to do most of your work from home.)
    * Strategy: pick projects wisely. *Drucker's Dictum: Doing things right is not as important as doing the right things.* The IBM "context switching v. productivity" study. (v. the life of a professor – enjoy grad school!)

- The conclusion
  - The lesson of the paper: don't just dive in and code and hope it all works out. Ruthlessly simplify!