“All problems in computer science can be solved by another level of indirection”

- Butler. W. Lampson

Lampson and his Hints

- prashanth
Overview

- About Lampson
- Xerox PARC
- Hints
Lampson -- Bio

1943 - born - Washington DC

1964 - AB (physics) - Harvard

1967 – PhD (ECE) UCB

1967-71 – Faculty (ECE) UCB & Berkeley Computer Corporation

1971-83 – Xerox Palo Alto Research Center

1984-95 – Systems Research Center, DEC. 1987- – Adjunct Prof, MIT

1992 – ACM Turing Award for “Contributions to the development of distributed, personal computing environments”

1995- Microsoft Corporation

1995- – Microsoft Corporation

1995- Microsoft Corporation
'My choice as the finest computer scientist of the century' – Bob Taylor who's known Lampson for 30 years

'You couldn't respond to his arguments, because the whole processing capacity of your brain was sent on trying to understand him!' -- Stuart Card at Xerox PARC

Eponym on Lampson!

milliLampson: /mil'@·lamp`sn/, n.

A unit of talking speed, abbreviated mL. Most people run about 200 milliLampsons. The eponymous Butler Lampson goes at 1000.

Ironically, Lampson after building 3 time-sharing systems (GENIE – UCB, BCC, MAXC – PARC) was first skeptical about personal computing (until MAXC's failure with graphic displays):

'We saw it as interesting from an intellectual point of view, not for building systems' – Lampson of UI and graphics
At Xerox PARC

The Big Four

Robert Taylor
Butler Lampson
Chuck Thacker
Alan Kay

Beanbag Room, Xerox PARC
Alto System
- first PC with bit-mapped graphics
- GUI with windows, icons, menus, mouse pointers
- Object oriented programming language
- WYSIWIG word processing
- laser printer
- Ethernet

Xerox Star
- bigger than Alto – sad failure!
- complex software
- closed monolithic system
- bad biz model
- “we certainly had the idea that computers like the star would be mass market items within a decade, but we just didn't pay attention to a business model of how to do it”

Soon enough the “gang” moved to SRC
Hints based on notion of an *interface* that separates an *implementation* of some abstraction from the *clients* who use the abstraction.

An interface is a contract to deliver a certain amount of service; clients of the interface depend on the contract, which is usually documented in the interface specification. Interface should be:

- simple
- complete
- admit a small and fast implementation
## Hints -- summary

<table>
<thead>
<tr>
<th>Why?</th>
<th>Functionality</th>
<th>Speed</th>
<th>Fault-tolerance</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Where?</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Completeness</td>
<td>Does it work?</td>
<td>Is it fast enough?</td>
<td>Does it keep working?</td>
</tr>
<tr>
<td>Interface</td>
<td></td>
<td>Shed load</td>
<td>End-to-end</td>
</tr>
<tr>
<td>Do one thing well:</td>
<td>Separate normal and worst case</td>
<td>End-to-end</td>
<td>End-to-end</td>
</tr>
<tr>
<td></td>
<td>Don’t generalize</td>
<td>Safety first</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Get it right</td>
<td>Make it fast</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Don’t hide power</td>
<td>Split resources</td>
<td>End-to-end</td>
</tr>
<tr>
<td></td>
<td>Use procedure arguments</td>
<td>Static analysis</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Leave it to the client</td>
<td>Dynamic translation</td>
<td>Make actions atomic</td>
</tr>
<tr>
<td></td>
<td>Keep basic interfaces stable</td>
<td>Cache answers</td>
<td>Make actions atomic</td>
</tr>
<tr>
<td></td>
<td>Keep a place to stand</td>
<td>Use hints</td>
<td>Use hints</td>
</tr>
<tr>
<td>Implementation</td>
<td>Plan to throw one away</td>
<td>Use brute force</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Keep secrets</td>
<td>Compute in background</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Use a good idea again</td>
<td>Batch processing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Divide and conquer</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1: Summary of the slogans
Hints in this...

- Categorized based on:
  - Simplicity & Generality
  - Level of Abstraction
  - Flexibility
  - Change and Continuity
  - Completeness
  - Speed-up
  - Division
  - Disaster Avoidance
  - Fault-tolerance
Do one thing at a time, and do it well -- generalizations are generally wrong

Seldom used interface can sacrifice some performance for functionality

Illustrations:
- PL/1: generalization => “unpredictable” cost
- Provide consistent meaning of large # of generic ops across wide variety of data-types
- Time with single-bit SWITCH: \[ #3 = 2 \times (#1 \text{ or } #2) \]
- Time with multi-bit SWITCH: \[ #2 = 2.5 \times (#1 \text{ or } #3) \]
#1 – Simplicity & Generality

- Tenex System: generalization => unexpected complexity
  - unassigned page reference => interrupt to user program
  - syscall unassigned page reference (also) => interrupt to user program
  - CONNECT syscall sends password
  - password of length 'n' (7-bit character) cracked in \((64 \times n)\) instead of \((128^n / 2)\) times!
  - manipulate a page ref fault for each position in the string by making that position as the last char in the page and next page is unassigned

CONNECT is implemented by a loop of the form

```plaintext
for i := 0 to Length(directoryPassword) do
    if directoryPassword[i] ≠ passwordArgument[i] then
        Wait three seconds; return BadPassword
    end if
end loop;
connect to directory; return Success
```
#1 – Simplicity & Generality

- Dorado Memory System: high implementation costs => only accounted by extensive experience in the past and wide usage in future
  - cache (64 ns cycle) and high-bandwidth (500 Mbits/second) fast path for I/O
  - 850 MSI chips and man-years of design
  - justified only if 30 yrs of prior usage or knowledge that memory access is bottleneck

- RISC vs. VAX: Make it fast rather than general and powerful
  - [simple & fast] vs. [general, powerful & slow]
  - VAX has powerful longer instructions
  - BUT, programs spend most time doing simple things: loads, stores, tests
  - for the same amount of hardware RISC faster than VAX!

- Word Processor: be(a)ware of abstraction – Get it right
  - FindNamedField() searches through doc (of length 'n') for the field
  - FindIthField() took O(n^2) instead of O(n) !!

```plaintext
for i := 0 to numberOfFields do
    FindIthField; if its name is name then exit
end loop
```
Don't hide power
- don't bury power with higher levels of abstraction
- purpose of abstractions => to conceal undesirable properties; desirable ones should not be hidden.

Keep Secrets of implementation
- conceal assumptions about implementation from client

Sometimes, increasing assumptions => improved performance
- a set of size 'n' – if known to be sorted, a membership test takes time log n rather than n!
- ability to improve each part separately is more important
#3 – Flexibility

- **Use procedure arguments**
  - illustrations:
    - Spy system monitoring facility in 940 system
    - installs patches coded in assembly by untrusted progs
    - checks for branches, loops, length, restricted store range
    - Bravo editor
    - Interval is a triple [document version, first character, last character]
    - FormattingOp is a function from properties to properties; a property might be italic or bold etc.

\[
\text{ChangeProperties}(\text{where: Interval, what: FormattingOp})
\]

- **Leave it to the client**
  - as long as it is cheap to pass control back and forth
  - if it works out fine => simplicity + flexibility + high-performance
  - illustration:
    - Monitors – Synchronization technique
      - provides only locking and signaling; leaves real work to client
      - clients pay only for what they need
      - lack of monitor support for scheduling of processes waiting on condition variables – illustrates this too
Keep basic interfaces stable
- break system into smaller pieces related only by stable (mostly OS/PL) interfaces

Keep a place to stand (if above not poss.)
- compatibility package – old on top of new
- illustration:
  - Tenex & Cal simulated supervisor calls of TOPS-10 & Scope resp.
  - extend this => Virtual Machines

Plan to throw one (an implementation) away
- redo totally rather than change the old one
- prototype! -- costs a lot less
- copy from previous one
- illustration:
  - Unix took ideas from Multics – even though the latter was too grandiose
  - revisit old decisions as system evolves => optimizations for certain load possible
Use a good idea again (instead of generalizing it)

- illustrations:
  - Star office system (windows, mac etc)
  - small set of ops (move, copy, delete) => on nearly all objects of the system
  - meaning => function of class of object
  - Data replication across machines for reliability
  - built over => transaction storage which again depends on => local replication for reliability!
#5 - Completeness

- **Handle normal and worst cases separately**
  - The normal case must be fast.
  - The worst case must make some progress.
  - Illustrations:
    - paging system
      - worst case => free page cushion disappears
      - have resources to free one item “under the mattress”
      - thus, worst case => one item freed at a time; slow => but progress!
    - deadlock handling
      - for better performance => don’t prevent/avoid it
      - “one crash a week is usually a cheap price to pay for 20% better performance” - still so?
Use Static analysis if you can
- don't delude yourself with a bad one; use dynamic
- illustration:
  - register allocation by compilers depend on availability of multiple register sets or stacking them during procedure calls

Use Dynamic translation
- from convenient (compact, easily modified or easily displayed) representation => one that can be quickly interpreted
- illustration:
  - Smalltalk runtime translates 'bytecodes' (from Smalltalk compiler) to machine language one procedure at a time
  - needs effective caching of translated code

Compute in Background when possible
- illustration:
  - writing out dirty pages
  - e-mail retrieval & delivery; “delivery within an hour or two is usually acceptable” - not so now :-)
  - synchronization between background & foreground process
Use Batch Processing if possible
- doing incrementally costs more
- secondary storage works better when sequentially accessed
- illustrations:
  - Bank of America's nightly batch runs – to update deposits and withdrawal; antiquated !?
  - today's eg.: tape archival ?

Cache answers
- to expensive computations (operations), rather than doing them over
- Store \([f, x, f(x)]\) – change or invalidate
- choose cache size adaptively
- illustration:
  - Virtual Memory [fetch, addr, data at addr]
  - Processor Cache
  - In a different setting – Bravo Editor
  - space for paragraph structure info (# of lines, margins, spacings etc) < space for storing those chars
  - pagination faster than page rendering
  - relies for performance on caching of [paragraph, shapeof(paragraph)]
Use hints to speed up normal execution

- cache vs. hints; hints:
  - may be wrong – thus, checked against “truth” before taking unrecoverable action
  - may not involve associative lookup

- illustrations:
  - Alto & Pilot File Systems
    - “truth” => file ID stored on each page of the file – leader page has file name and directory
    - “hint” => directory containing (file name, file ID, leader page address)
    - to access a file with a name, first use the “hint” => then hint verified with truth before usage => if wrong, scan the whole disk and reconstruct it

- Store and Forward routing first used in Arpanet
  - “hint” - broadcast reports about neighbour states
  - “truth” - I know my identity

- Ethernet packet sending – lack of carrier signal
- Branch Prediction
#7 – Division

**Divide and Conquer**
- when resources are limited -- bite off as much as will fit; leave the rest for the next iteration
- sometimes better to split resource in quantized fixed size units
- illustrations:
  - Virtual Memory Pages vs. Variable Size Segments
  - Dover Raster printer
    - scan-convert 3300x4200 (14 million) array of bits
    - too large for memory & disk access slower than printing rate
    - printer electronics => 2 band buffers 16x4200 => one to accept incoming stream while the other scan converts => stream divided in these chunks

**Split resources** in a fixed way if in doubt, rather than sharing them
- dedicated resources across entities – faster access – but more total resources
- illustrations:
  - faster to access info in registers than from memory -- “even if the machine has a high-performance cache” ! - now ?
  - Interlisp virtual memory system had virtual-physical mappings stored in “real memory” to avoid circularity complexity !
Safety first
- in allocating resources, strive to avoid disaster rather than to attain an optimum
- general purpose system cannot optimize resources -- rather than optimize provide more capacity
- illustrations:
  - Memory paging
    - swapping optimization to squeeze out every byte NOT required
    - memory is cheap – simple scheme with demand paging works fine -- only worry about thrashing or too much demand for memory
  - Processor Scheduling
    - intelligent schemes failed – only basic priority (interactive vs. non-interactive; background vs foreground) schemes worked

Shed Load
- to control demand shed load rather get overloaded
- illustrations
  - Bob Morris' “large red button” -- let people not waste time in front of terminals
  - Arpanet's guaranteed packet deliveries – Pup discarded ruthlessly !
  - Memory Manager limits jobs
End-to-end error recovery is absolutely necessary for a reliable system

- other error detection and recovery is NOT logically necessary – only for performance
- problems – requires cheap tests & performance defects
- illustrations:
  - Pup Internet
    - “best efforts” delivery
    - packet loss – wrong routing, router buffer overflow etc..
    - end-to-end recovery must – by client
    - packet transport may report problems (discarded packets, checksum mismatch) – these only improve performance does NOT replace end-to-end

Log Updates

- to record the “truth” about the state of an object
- append-only storage – cheap I/O – reliable in the face of crashes – replicated for redundancy
- current state of the object => “hint”
- log entry – (name of update procedure, args)
- Requirements:
  - Update procedure => true function (func only of args, no side-effect on other objects)
  - Arguments => immediate or immutable (versioned, if changing) objects
...#9 Fault-tolerance

- illustration:
  - Bravo Editor
    - Interval is a triple [document version, first character, last character]
    - FormattingOp is a function from properties to properties; a property might be italic or bold etc
    - only two kinds of log entries

Replace(old: Interval, new: Interval)
ChangeProperties(where: Interval, what: FormattingOp)

- Make actions atomic or restartable
  - atomic action => either completes or has not effect
  - illustration:
    - In database systems (using Logs):
      - id per atomic action – tag log entries with that action id – commit record
      - after failure – redo each committed action – undo aborted actions
      - log entry => “idempotent”
  - non-trivial in general, but sometimes a weaker cheaper method will work
  - illustration:
    - Grapevine’s replicated database of [names, distr lists]
      - time-stamped updates sent through mails only
      - delays not important to usefulness of the system here
Finally..

- Written two decades back, most of these design principles hold good even today!
- Does this say anything about Systems Research in this time or in the future??