LucyBookClub

Lucy Book Club

Schedule and Location

The Lucy Book Club will be taking a hiatus through ApacheCon Denver.

We ordinarily meet weekly on Tuesday evenings at 18:00 PDT in a Google Hangout.

Log into Google Plus, then join the conversation.

We used to meet at 19:00 PDT, but we have changed the schedule to be more convenient for our East Coast participants.

Agenda

The Lucy Book Club is open to anyone who is interested in search, parsing, compilers, and Lucy in general.

We are currently reading through the book Pro Git by Scott Chacon. The entire book is available online as well as in traditional dead-tree format.

Suggested additional reading for the intrepid: the gitglossary(7) man page, Build git - learn git

Where To Get Information

irc.freenode.net/#lucy_dev

dev@lucy.apache.org

Upcoming Meetings

None scheduled.

Community Notes

Below are an excerpt of notes from previous meetings and or anything of interest related to the meetings.

Tuesday March 11, 18:00 PDT:

Pro Git Chapter 9: Git Internals

Discussion questions:

1. What is a content-addressable file system?
2. After `git init`, the `.git` directory contains 8 entries. What are they and what purpose does each serve?
3. What does the file `.git/HEAD` contain when the repository is in a "detached HEAD" state?
4. In Git jargon, what's the difference between "plumbing" and "porcelain"?
5. What do `git write-tree` and `git read-tree` do?
6. Describe a recipe for performing a sequence of commits using only plumbing commands.
7. What's the difference between how a lightweight tag and an annotated tag are represented in the Git file system?
8. How can you tag a blob?
9. Why use `update-ref` and `symbolic-ref` rather than write content directly?
10. What is the file format of every Git object?
11. What are the valid file modes for a blob?
12. What information does each entry in a tree contain?
13. Where are remote branches stored?
14. Describe the circumstances under which Git may remove unreferenced objects.
15. Why keep the most recent revision as a complete blob in a pack file, rather than the first revision?

Tuesday March 4, 18:00 PDT:

Pro Git Chapter 7: Customizing Git

Discussion questions:

1. What are some settings you might want to customize via `git config --global`?
2. How can you change the default Git commit message?
3. Why not set `color.ui` to `always`?
4. What two whitespace checks are enabled by default?
5. How do you disable non-fast-forward pushes for the entire repo? How about only for the `master` branch?
6. What hole does the server setting `recieve.denyDeletes` plug?
7. Why might you turn on `receive.fsckObjects`?
8. How can you persuade Git to show you a diff between different versions of an MS Word file?
9. Conceptually, what might a diff of JPEG file show?
10. How do you manage format options for `git archive`?
11. How do you install a hook?
12. What are some situations where you'd use a client-side hook? How about a server-side hook?
13. How would you set up Git to run code files through a prettifier before committing using attributes and a filter? Wouldn't it be nice if it just warned instead of silently modifying your files? How could you set up that behavior using a pre-commit hook?
14. How can you insert auto-generated content into a commit message?
15. What's the downside of using client-side hooks to enforce policy? Are server-side hooks better?
16. What mechanism do hooks generally use to abort the process that invoked them?
17. What server-side hook would you use for a commit mailer?

Tuesday February 25, 18:00 PDT:

Pro Git Chapter 6: Git Tools

Discussion questions:

1. What's a short SHA?
2. What's more likely, a SHA collision, or having every single member of your programming team killed by wolves in separate incidents tonight?
3. What tool will tell you the SHA that a commit specifier aliases?
4. What's the difference between `HEAD~2` and `HEAD^2`? What does `HEAD~3^2` mean?
5. How can you check what you're about to push?
6. What's the difference between `...` in `git log` and `git diff`?
7. If you have staged changes, what is the effect of `git stash` followed by `git stash apply`?
8. Scott Chacon says that the material in this chapter is stuff we won't use every day. However, `git commit --amend` is covered. Is Scott nuts?
9. Describe the process of splitting an existing commit during interactive rebasing.
10. Which is easier to use, `git add -i` or `git add --patch`?
11. What two commands can be used to simulate `git stash pop`?
12. Describe a use case for `git stash branch BRANCHNAME`.
13. During an interactive rebase, what do the following commands do? pick, reword, edit, squash, fixup, exec.
14. How would you break a directory within a repo out into its own repo, complete with history? If that directory has moved around, is it possible to preserve its history?
15. How would you use `git bisect` to find the revision where a bug originated using a script? How about interactively?
16. Why are submodules referenced using SHAs rather than branch names?

Tuesday February 11, 18:00 PDT:

Pro Git Chapter 5: Distributed Git

Discussion questions:

1. What error message does Git give you when you try to push something not up-to-date?
2. Describe the diffence between the "Integration Manager Workflow" and the "Director and Lieutenants Workflow".
3. What's the difference between `git apply` and the Unix utility `patch`?
4. Does `git apply` commit?
5. Who is the Committer after `git am` is used to apply patch sequences generated with `git format-patch`?
6. Why is it necessary to rebase the `develop` branch on top of `master` periodically?
7. What are the four branches used by the Git project? How often are they rebased?
8. How is cherry-picking related to rebasing?
9. What is the `--not` flag to `git log`?
10. Why would you use the triple dot to git log? Why is order important?
11. When would you use `git merge --no-commit`?
12. What is the recommended format for commit messages, per the docs for `git commit`?
13. What is the default format of the output generated by `git describe`?
14. What are some typical guidelines for submitting patches?
15. Why do many projects insist that you rebase any pull requests, or possibly squash everything in the branch down to one commit?

Tuesday February 4, 18:00 PDT:

Pro Git Chapter 4: Git on the Server

Discussion questions:

```
Read:

    4. Git on the Server
    4.1 The Protocols
    4.2 Getting Git on a Server
    4.3 Generating Your SSH Public Key
    4.4 Setting Up the Server
    4.5 Public Access
    4.6 GitWeb

Skip:

    4.7 Gitosis
    4.8 Gitolite
    4.9 Git Daemon

Skim:

    4.10 Hosted Git
    4.11 Summary
```

Discussion questions:

1. Technically, there is no need for a master repository with distributed version control a la Git – collaborators can just share code directly. Why doesn't anybody do things that way? 2. Why are repositories set up to be accessed remotely generally "bare" repositories? 3. What four protocols does Git support for remote access to a repo? 4. For a local repo, what changes when you specify the `file://` protocol to `git clone` instead of just supplying a path? 5. Why not use the `git` protocol for write access? 6. What file needs to be present in a repo in order to enable access

via the `git` protocol? Bonus question: Why do you suppose that Git's authors imposed this requirement? 7. Which is the "dumb protocol", which is the "smart protocol", and what distinguishes them from each other? 8. If you have no collaborators, is there any downside to using the SSH protocol? 9. What naming convention is traditional for bare repositories? 10. When setting up a server for SSH access and the `git` user, where do the SSH public keys go? 11. Why is interacting with a Git repo on an NFS mount slow? 12. What "hook" do you have to set up in order to facilitate HTTP access to a repo? What steps are necessary to enable it? What does it do? 13. If you have a Mac with Git installed, try running `git instaweb --httpd=webrick` on a local repo. What happens if you edit the file `.git/description`? (When you're done, stop the server by running `git instaweb --httpd=webbrick --stop`.) 14. Check out the drop-down list of search options in the GitWeb interface. What are the options? Bonus question: what do they do? 15. What are the pros and cons of using a Git hosting service like GitHub instead of running your own Git server?

Tuesday January 28, 18:00 PDT:

Pro Git Chapter 3: Git Branching

Discussion questions:

1. What is the content of a Git commit object?
2. What is a branch in Git?
3. What happens to the branch you're on when you commit? Does `.git/HEAD` change?
4. What's the difference between `git branch foo` and `git checkout -b foo`?
5. What are the different kinds of refs stored in `.git/refs`?
6. What happens when you perform a `git checkout`?
7. How does Git handle collisions between tracked and untracked files during `checkout`?
8. What is a `fast forward` merge?
9. After a `git clone`, how is `master` set up for you?
10. How do you commit to a remote branch?
11. Why does a preparatory `rebase` make it easier to accept and apply a patch?
12. When resolving a merge conflict, what does `git status` display? How about after the conflict is resolved?
13. What does it mean to have a common ancestor? What does the book mean when it says that Git selects the common ancestor automatically, while the user must select it in other version control systems?
14. What's the difference between the lowercase d and uppercase D switches to `git branch`?
15. Describe a three-tier hierarchical development branch system.
16. Is it a good idea for the local branch to have a different name than the remote branch it's tracking?
17. What happens when you rebase?

Tuesday January 21, 18:00 PDT:

Pro Git Chapter 2: Git Basics

Discussion questions:

1. What does `git init` do?
2. What data is not copied via `git clone`?
3. What does it mean when `git status` reports "(working directory clean)"?
4. What does it mean when a file is listed under both `Changes to be committed` and `Changes not staged for commit`?
5. I know I've changed something. Why isn't `git diff` showing me anything?
6. What are some of the functions served by `git add`?
7. How is `git log -p` different from `git show`?
8. How do you skip the staging area when committing?
9. What's the opposite of `add`?
10. How do you unstage a file without blowing away changes?
11. How do you blow away changes?
12. How is `git pull` different from `svn up`?
13. What is `git mv` a shortcut for?
14. What's the difference between author and committer? How do you display both in `git log` output?
15. What do you do when you've screwed up the last commit?
16. How do you persuade `git log` to show *which* files changed?
17. How do you display the branch visuals with `git log`?
18. What's the difference between a lightweight tag and an annotated tag? How are lightweight tags like branches?
19. What does git auto-completion get you? How is it implemented?

Tuesday January 14, 18:00 PDT:

Pro Git Chapter 1: Getting Started

Discussion questions:

1. What are some backup strategies that don't use version control?
2. What are some of the features of version control?
3. What does RCS stand for?
4. How does RCS produce the current contents of a file which has been changed many times?
5. How is RCS better than saving copies of files in time-stamped directories?
6. How does version control differ from a backup system like Time Machine?
7. What problem inherent to local version control systems like RCS were centralized version control systems designed to solve?
8. What are the pros and cons of centralized version control?
9. What are the pros and cons of distributed version control?

10. Which version control systems store snapshots and which store differences?
11. How did Linux manage version control until around 2002?
12. What were some of the design criteria for Git?
13. Why is `git log` so much faster than `svn log`?
14. Would Git work well for the following tasks?
    - writing a novel
    - writing poetry
    - writing a journal
    - writing magazine articles
    - taking notes on academic classes
    - managing a small static website
    - form letters
    - archiving log files
    - documentation
    - magazine layouts
    - raster image files, e.g. Photoshop native format
    - .m3u playlists
    - video editing projects
    - MIDI music sequence files
15. How does addressing content by hash improve Git's integrity?
16. What are the security advantages of a DVCS?
17. When your commit changes only one file out of many, how does Git handle saving the state of the files which have *not* changed?
18. What are the three states that a tracked file can be in at any time? What is the state of an *untracked* file?
19. What is Git's "staging area" and how is it implemented?
20. What are the three levels of config files for Git? Which overrides which?
21. When you run `git config --list`, you may see the same key multiple times with different values. What does this mean?

Tuesday June 18, 7:00 pm PDT:

Lectures from The Hardware/Software Interface

- Section 9: Virtual Memory
- Section 10: Memory Allocation

Review questions for section 9, "Virtual Memory":

1. What four problems do the professors suggest that virtual memory solves?
2. "Any problem in computer science can be solved by..." what?
3. Without indirection, what happens if you need to update data which exists at multiple memory locations?
4. What are some non-computer-science examples of indirection?
5. What aren't there any machines which are capable of mapping a full 64-bit address space in physical memory?
6. What are some systems which use direct physical addressing?
7. What are the three states for virtual memory pages?
8. Roughly how large is a virtual memory page?
9. Why not use write-through with virtual memory?
10. What's in a page table entry?
11. How many page tables per process?
12. How much of the MMU is implemented with hardware?
13. What's *thrashing*?
14. What permission bits are contained in typical page table entries?
15. Explain the sequence of events during a page hit.
16. What impact does address space size have on page table size?
17. How is responsibility divided between hardware and OS for implementing virtual memory?

Review questions for section 10, "Memory Allocation":

1. How does the heap grow?
2. What alignment guarantees does malloc make?
3. What are the two conflicting performance goals of memory allocators?
4. There are two kinds of fragmentation: internal and external. Internal and external to what? How do they differ?
5. Briefly describe four approaches to managing a memory heap: *implicit free list*, *explicit free list*, *segregated free list*, and *blocks sorted by size*.
6. What are the differences between *first fit*, *next fit* and *best fit*?
7. What is *splitting*?
8. What is *coalescing*?
9. When memory is nearly full, which is faster: implicit or explicit free list?
10. For a segregated free list allocator, how closely does memory utilization approach best fit?

Bonus questions not answered by lecture:

1. If heap memory were executable, how might a double free() lead to a an exploit resulting in arbitrary code execution? Hint: it's like a stack buffer overflow, but much harder for the attacker.
2. Why does Professor Ceze keep saying "Simple, right?" for stuff that isn't simple?

Tuesday June 11, 7:00 pm PDT:

Lectures from The Hardware/Software Interface

- Section 8: Processes

Review questions:

1. Jumps and calls suffice for responding to changes in program state. What kinds of state change do they not suffice for? 2. What's the difference between synchronous and asynchronous exceptions? 3. Describe the three types of synchronous exceptions, and the differences between them in terms of continued program flow. 4. When does the "interrupt pin" get set? 5. Describe program flow on a *page fault* which results in a successful load from virtual memory. 6. What's the difference between a program and a process? 7. What are the two key abstractions provided by processes? 8. Describe how the OS kernel uses exceptional control flow to provide the illusion of continuous execution to multiple concurrent processes. 9. What is the distinction between *concurrent* and *parallel*? 10. Calling `execve` replaces a process's address space and code with a new executable. What state, if any, persists across a call to `execve`? 11. What order do parent and child processes execute in after a fork()? 12. What does execve() return? 13. Where are the parameters passed via execve() stored after the call completes? 14. What are the consequences if a long-running parent process fails to reap zombie children? What happens once the neglectful parent terminates?

Bonus questions not answered by lecture:

1. Why does output to the standard file descriptors appear to be ordered when when processes exit()? 2. How expensive is fork()? 3. Why use fork /execve instead of posix_spawn? How about vfork/execve as an alternative?

Tuesday June 4, 7:00 pm PDT:

Lectures from The Hardware/Software Interface

- Section 7: Memory and Caches

Review questions:

1. How well has memory speed and memory bandwidth kept up with increases in CPU speed?
2. What is the unit of memory which is transfered between caches?
3. What are the two types of *locality*?
4. How should code for a loop which iterates over a multi-dimensional array change depending on whether the array is stored in column-major or row-major order?
5. Why is it faster to perform matrix multiplication in blocks rather than row by row?
6. How much faster do hits have to be than misses in order for a 99% hit rate to be twice as fast as a 97% hit rate?
7. Why is *miss rate* often used rather than *hit rate*?
8. What's the typical miss penalty for L1? L2? Main memory? Can there be a miss penalty for data stored a register?
9. How are caches divided within the Intel Core i7?
10. What are the drawbacks of a *direct-mapped* cache? What are the drawbacks of a *fully associative* cache?
11. What purpose does the *tag* in a cache entry serve?
12. In addition to the tag, a cache read also involves a *set index* and a *block offset* – but those numbers are not stored explicity as part of the cache entry. How are they used?
13. How is the *valid bit* used in the context of a write-back cache?
14. What's are the differences between a "cold miss", a "conflict miss" and a "capacity miss"? Is it possible to have a "conflict miss" in a fully associative cache?
15. What data was used to generate the "Memory Mountain" graphic on the front cover of the text?

Bonus questions not answered by lecture:

1. What's faster: SRAM or DRAM? Why?
2. Describe situations in which spacial locality can be exploited – both for data and for instructions.
3. According to the professors, the "big idea" of the memory hierarchy is to create "a large pool of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top." What usage pattern might defeat this technique?
4. Data intensive applications (such as large-scale search engines) may be constrained by memory bus bandwidth despite caching. What are some techniques to address this problem?

Tuesday May 21, 7:00 pm PDT:

Lectures from The Hardware/Software Interface

- Section 5: Procedures and Stacks
- Section 6: Arrays and Structs

Review questions:

1. In the Linux process memory layout, the instructions and stack are at different ends of the address space. Which is on the high end, and which is on the low end? 2. Under normal circumstances, what portions of memory can contain executable instructions? (stack, heap, static data, instructions) 3. What is a return address? How is it stored when a procedure is called, so that the callee can get at it? 4. In what ways are the `call` and `ret` instructions complementary? 5. Why might registers be divided between caller-saves and callee-saves, rather than being all one or the other? 6. Where is the standard location for returning a value? 7. Without a stack, only global variables are available for use by subroutines. What popular language feature would this cripple? 8. Does the callee always return before the caller under all possible circumstances? No exceptions? 9. What gets stored in a stack frame? How much less gets stored on the stack in x86-64 than in IA32? 10. Imagine a function which takes 8 arguments of type `int` named `arg1` through `arg8`. How would they be passed under IA32? Under x86-64? Under IA32, what will you find at `8(%ebp)`? How about at `4(%ebp)`? 11. What's the difference between a multi-dimensional array and a multi-level array? Which one is used by Java? 12. When the compiler compiles a loop over a fixed range of elements in an array, it may use pointer math rather than indexing. Why might this be more efficient? 13. Are C nested arrays guaranteed to be implemented using "row-major" or "column-major" order? 14. How can a single `leal` instruction be used to multiply the value in a register by 5? 15. What is the definition of "aligned data"? 16. Why is it inefficient (at best) to retrieve unaligned data? 17. What's the difference between IA32 and 32-bit Windows with regards to alignment of `double`? 18. What are some of the countermeasures which have made stack-smashing attacks more difficult?

Bonus questions not answered by lecture:

1. Under x86-64, the frame pointer (%ebp in IA32) is omitted. How will everything be addressed relative to the stack pointer when alloca() is invoked? 2. In C, `array[2]` and `2[array]` are equivalent. So are `array[-3]` and `-3[array]`. Why? 3. Does `sizeof` include padding bytes at the end of a struct in its calculation? 4. Is a non-executable stack in a JIT environment still helpful?

Tuesday May 14, 7:00 pm PDT:

Lectures from The Hardware/Software Interface

- Section 3: Basics of Architecture, Machine Code
- Section 4: x86 Assembly

Review questions:

1. What are the four steps of C compilation? 2. What's the difference between *instruction set architecture* and *microarchitecture*? Under whose domain is cache size? Core frequency? Number of registers and their width? 3. What processor introduced "flat addressing"? 4. What are the four registers in x86 and x86-64 which describe condition codes? What purpose do they serve? 5. What are the three basic categories of assembly instructions? 6. How typesafe is assembly code? 7. What will disassembling a function with `gdb` tell you that disassembling with `objdump` will not? 8. What are the origins of the names of the x86 registers? 9. How would you get at a single byte within an x86 register? 10. What are the four integer op suffixes from AT&T assembler and what do they stand for? 11. What is the pointer dereferencing operator in AT&T assembler? What character is used as a sigil for constants? 12. Describe *displacement* and *indirect* addressing modes. Describe the "general form" of the addressing idiom. 13. How can the LEA instruction be abused to perform simple arithmetic using the general form of addressing? What checks are omitted by LEA which are performed by standard arithmetic instructions like ADD? 14. What allows chips which implement the x86-64 ISA to perform fewer manipulations of the stack than x86 chips? 15. What's the difference between the SARL and SHRL instructions? 16. What register do the jX "jump" instructions modify? 17. What do testX and cmpX do? 18. What it is the point of `testl %eax %eax`? 19. Which control construct requires more JMP instructions: while or do-while? How many JMP instructions does a for-loop require? 20. How does the `cmovC` "conditional move" instruction make it possible to avoid a JMP? 21. What advantage does PC-relative addressing offer over absolute addressing? 22. Under what circumstances can a `switch` statement be implemented as a "jump table"? 23. Explain the instruction `jmp *.L62(,%edx,4)`

Tuesday April 30, 7:00 pm PDT:

Lectures from The Hardware/Software Interface

- Section 0: Introduction
- Section 1: Memory, Data and Addressing
- Section 2: Integer and Floating Point Numbers

Review questions:

1. What is hardware? What is software? What is the "hardware/software interface" as defined by the instructors? What do you, personally, hope to gain from studying the "hardware/software interface"? 2. What attributes of assembly language make it easier to understand than machine code? 3. Why might a computer process/thread be described as an "illusion"? 4. What aspect of the low and high voltages representing zeroes and ones limits CPU clock speed? 5. In what sense are CPU registers, CPU cache, main memory, persistent storage (e.g. hard disk), and offline storage (e.g. offsite backups) all "memory"? How do they differ? 6. What are the advantages of little-endian architecture with respect to casting between integer widths? 7. Adding 1 to a pointer adds how much to the address? 8. In a C assignment statement, what must the left-hand side evaluate to, and what must the right-hand side evaluate to? What do scalar variable names in C represent? 9. What determines the value of the expression `&array[1] - &array[0]`? 10. How would you implement a `show_bits` function to complement the `show_bytes` function from the lecture on arrays? 11. What are the practical applications of [DeMorgan]'s Law? 12. Why doesn't C provide support for bit arrays, instead only supporting the application of "bitwise" operators to integral data types? There has to be a better way... right? 13. In C, how do the logical complement operator and the bitwise complement operator differ? 14. What set operations do the four bitwise operators `& | ^ !` correspond to? 15. What is a *one-hot* encoding? A *two-hot* encoding? 16. What are the drawbacks of *sign-and-magnitude* representation of negative numbers? 17. In a 8-bit *two's-complement* representation, what is the value of the LSB? Of bits 2-7? And finally, what is the value of the MSB? 18. Which representation allows for reuse of the adder used for unsigned integers on the CPU chip: one's complement, or two's complement? 19. How can an adder be adapted to perform binary subtraction? 20. When casting, what operations does the processor actually perform? In what cases is casting just a matter of reinterpreting a bit pattern which exists in memory, and in what cases does the bit pattern have to be changed? 21. What is the result of the C expression `-1 < 0U`? 22. Of the problems you solved in bits.c programming assignment, which was the most satisfying and why?

Tuesday April 23, 7:00 pm PDT (Postponed from April 16th):

Drepper Paper:

```
    2.5 Improving Generated Code
    2.6 Increasing Security
    2.7 Simple Profiling of Interface Security
    3.0 Maintaining APIs and ABIs
    3.1 What are APIs and ABIs?
    3.2 Defining Stability
    3.3 ABI Versioning
    3.4 Restricting Exports
    3.5 Handling Compatible Changes (GNU)
    3.6 Handling Compatible Changes (Solaris)
    3.7 Incompatible Changes
    3.8 Using Versioned DSOs
    3.9 Inter-Object File Relations
```

Review questions:

1. For IA-32, how can one help reduce access to the GOT address? Why is this a bad idea for IA-64?

2. The -z relro linker option will help fix what security issue with DSOs? How would one turn off lazy relocation?

3. Re-review, what is the PLT stand for again? And how is it different from the GOT?

4. What happens if one is profiling and the PLT is not used? What would the DL_CALL_FCT macro do?

5. What is the ABI?

6. What does LD_BIND_NOW do?

7. What was Sun's solution to versioning and how does it work?

8. Why should symbol maps be used all the time?

9. Take a look at the code sample on page 37, left hand column, what is the .symver doing this instance?

10. What does @@ mean? What does just @ mean?

11. In a mapping file what does local: * mean?

12. What does LD_LIBRARY_PATH do? What are run paths?

13. What is a DST and how are they useful?

Tuesday April 9, 7:00 pm PDT:

Drepper Paper:

```
        2.2.4 Define Visibility for C++ classes
        2.2.5 Use Export Maps
        2.2.6 Libtool's -export-symbols
        2.2.7 Avoid Using Exported Symbols
    2.3 Shortening Symbol Names
    2.4 Choosing the Right Type
        2.4.1 Pointers vs Arrays
        2.4.2 Forever Const
        2.4.3 Array of Data Pointers
        2.4.4 Array of Function Pointers
        2.4.5 C++ Virtual Function Tables
```

Review questions:

1. What purpose does -fvisibility-inlines-hiddens serve, especially in regard to C++?

2. Why is the strategy of using the most restrictive visibility possible a good practice when writing C++ to work with Elf?

3 What is a symbol map file for?

4 What is a wrapper function? And what are drawbacks of using wrapper functions?

5. What is an alias?

6. Is using DF_SYMBOLIC a good idea?

7. Why is the length of symbol name important? And why in C++ is this a problem?

8. Why is char str[] = "some string" more optimal than char *str = "some string"?

9 What are some other ways to make string lookup faster?

10. Why is using read only memory when possible more optimal?

11 Why is a switch statement a good way to implement an array of functions like behavior?

12. How do virtual function tables impact startup time?

Tuesday April 2, 7:00 pm PDT:

Drepper Paper:

```
2.0 Optimizations DSOs
2.1 Data Definitions
2.2 Export Control
    2.2.1 Use static
    2.2.2 Define Global Visibility
    2.2.3 Define Per Symbol Visibility
```

Review questions:

Questions:

1. When compiling code that is going to be used as a DSO, why is it important to use the -fpic/-fPIC flags?

2. On page 16, right hand column, there are two code samples. What is different about them? And why is the second one significant?

3. What is the easiest way to not export a variable or function?

4. Why did the author add "static" to last and next in the first code sample on page 18, left hand column?

5. Since C doesn't provide a way to define visibility of a function or variable, how does GCC allow the programmer to indicate visibility within the code?

6. What does

7. If -fvisibility=hidden is used, how would I enable a symbol to have default visibility?

Tuesday March 19, 7:00 pm PDT:

(Rescheduled from March 12.)

```
    15.1.2 The Common Language Infrastructure
15.2 Late Binding of Machine Code
    15.2.1 Just-In-Time and Dynamic Compilation
    15.2.2 Binary Translation
    15.2.3 Binary Rewriting
    15.2.4 Mobile Code and Sandboxing
15.3 Inspection/Introspection
    15.3.1 Reflection
    15.3.2 Symbolic Debugging
    15.3.3 Performance Analysis
15.4 Summary and Concluding Remarks
```

Tuesday March 5, 7:00 pm PDT:

From PLP3:

```
15.1 Virtual Machines
    15.1.1 The Java Virtual Machine
```

Review questions: 1-8 on page 784.

From How To Write Shared Libraries by Ulrich Drepper:

```
    1.5 Startup in the Dynamic Linker
        1.5.1 The Relocation Process
        1.5.2 Symbol Relocations
        1.5.3 The GNU-style Hash Table
        1.5.4 Lookup Scope
        1.5.5 GOT and PLT
        1.5.6 Running the Constructors
    1.6 Summary of the Costs of ELF
    1.7 Measuring ld.so Performance
```

Review questions for the Drepper:

6. Why do NUL-terminated strings make horrible hash keys?
7. How can *interposition* be used to override the behavior of a named procedure?
8. How can different C++ name mangling schemes have an impact on symbol hash lookup times?
9. Compare ELF hash lookup with GNU-style hash lookup.
10. Consider this source code, which consists of a declaration of a function `munge` and an int `foo` which must be defined elsewhere, and a function `munge_foo` which references both.

```
        int
        munge(int num);

        extern int foo;

        int
        munge_foo(void) {
            return munge(foo);
        }
```

Narrate the following assembly generated for `munge_foo` when it is compiled as position-independent code.

```
        movl   foo@GOT(%ebx), %eax
        pushl  (%eax)
        call   munge@PLT
```

11. Summarize the costs of the GOT (Global Offset Table) and the PLT (Procedure Linkage Table).

Tuesday February 12, 7:00 pm PDT:

From PLP3:

```
    14.4 Address Space Organization
    14.5 Assembly
        14.5.1 Emitting Instructions
        14.5.2 Assigning Addresses to Names
    14.6 Linking
        14.6.1 Relocation and Name Resolution
        14.6.2 Type Checking
    14.7 Dynamic Linking*
        14.7.1 Position-Independent Code*
        14.7.2 Fully Dynamic (Lazy) Linking*
    14.8 Summary and Concluding Remarks

* includes auxiliary CD material
```

Additionally, we'll cover the first few sections of Ulrich Drepper's paper, How To Write Shared Libraries:

```
    1 Preface
        1.1 A Little Bit of History
        1.2 The Move To ELF
        1.3 How Is ELF Implemented?
        1.4 Startup: In The Kernel
```

Here are review questions for the Drepper material:

1. What limitations of the `a.out` format make it ill-suited for creating shared libraries? If a modern Linux system supplied shared libraries derived from `a.out`, how would that affect applications which use them? 2. What is the main difference between a compiled ELF shared library and a compiled binary executable? 3. If you're creating a very large application which takes a long time to link, how can you use shared libraries to minimize the edit-compile-test loop and maximize programmer efficiency? Compare this use of shared libraries to traditional separate compilation. 4. When loading the contents of an executable file into memory, why is it desirable to mark as many pages as possible non-writable? 5. Where must the `Elf32_Phdr` and `Elf64_Phdr` program header structs be located in an ELF object file? What is the first member in these program header structs?

Tuesday February 5, 7:00 pm PDT:

```
    14.1 Back-End Compiler Structure
        14.1.1 A Plausible Set of Phases
        14.1.2 Phases and Passes
    14.2 Intermediate forms*
        14.2.1 Diana*
        14.2.2 The gcc IFs*
        14.2.3 Stack-Based Intermediate Forms
    14.3 Code Generation
        14.3.1 An Attribute Grammar Example
        14.3.2 Register Allocation

* includes auxiliary CD material
```

Tuesday January 29, 7:00 pm PDT:

```
    13.3 Scripting the World Wide Web
        13.3.1 CGI Scripts
        13.3.2 Embedded Server-Side Scripts
        13.3.3 Client-Side Scripts
        13.3.4 Java Applets
        13.3.5 XSLT
    13.4 Innovative Features
        13.4.1 Names and Scopes
        13.4.2 String and Pattern Manipulation
        13.4.3 Data Types
        13.4.4 Object Orientation
    13.5 Summary and Concluding Remarks
```

Tuesday January 22, 7:00 pm PDT:

```
    13.1 What Is a Scripting Language?
        13.1.1 Common Characteristics
    13.2 Problem Domains
        13.2.1 Shell (Command) Languages
        13.2.2 Text Processing and Report Generation
        13.2.3 Mathematics and Statistics
        13.2.4 "Glue Languages" and General-Purpose Scripting
        13.2.5 Extension Languages
```

Tuesday January 15, 7:00 pm PDT:

The Lucy Book Club is taking a break from our book-in-progress this week to read a paper on integer compression techniques. One of the algorithms described in the paper is PFOR-DELTA (Patched Frame-Of-Reference with delta encoding), which is particularly suitable for inverted lists.

Super-Scalar RAM-CPU Cache Compression by Marcin Zukowski, Sándor Héman, Niels Nes, Peter Boncz

We'll go over the following questions:

1. Why is PFOR-DELTA interesting to Lucy? 2. What is a *segment* in PFOR-DELTA? 3. What are the 4 major parts of a segment? What is in them? 4. What are the tradeoffs for choosing different sizes of b (bit width)? 5. What type of data structure is used to keep track of exceptions? 6. What is a *compulsory exception*? How does it influence your choice of b? 7. PFOR-DELTA is a very interesting compression technique, but why is it really faster? What is PFOR-DELTA really optimizing for? 8. In their testing, was RAM-RAM or RAM-Cache faster and why? 9. Fine-grained access has some extra cost – what is it?

Tuesday January 8, 7:00 pm PDT:

```
12.5 Message Passing
    12.5.1 Naming Communication Partners
    12.5.2 Sending
    12.5.3 Receiving
    12.5.4 Remote Procedure Call
```

Thursday January 3, 7:00 pm PDT:

```
12.4 Language-Level Mechanisms
    12.4.1 Monitors
    12.4.2 Conditional Critical Regions
    12.4.3 Synchronization in Java
    12.4.4 Transactional Memory
    12.4.5 Implicit Synchronization
12.5 Message Passing
12.6 Summary and Concluding Remarks
```

Tuesday December 18, 7:00 pm PDT:

```
12.3 Implementing Synchronization
    12.3.1 Busy-Wait Synchronization
    12.3.2 Nonblocking Algorithms
    12.3.3 Memory Consistency Models
    12.3.4 Scheduler Implementations
    12.3.5 Semaphores
```

Tuesday December 11, 7:00 pm PDT:

```
12.1 Background and Motivation
    12.1.1 The Case for Multithreaded Programs
    12.1.2 Multiprocessor Architecture
12.2 Concurrent Programming Fundamentals
    12.2.1 Communication and Synchronization
    12.2.2 Languages and Libraries
    12.2.3 Thread Creation Syntax
    12.2.4 Implementation of Threads
```

Tuesday December 4, 7:00 pm PDT:

```
11 Logic Languages [the entire chapter -- dead-tree material only]
```

Tuesday November 27, 7:00 pm PDT:

```
    9.5 Multiple Inheritance [including aux material on CD and questions]

    10.4 Evaluation Order Revisited
        10.4.1 Strictness and Lazy Evaluation
        10.4.2 I/O: Streams and Monads
    10.5 Higher Order Functions
    10.6 Theoretical Foundations [book only]
    10.7 Functional Programming in Perspective
    10.8 Summary and Concluding Remarks
```

Tuesday November 13, 7:00 pm PDT:

```
    9.6 Object-Oriented Programming Revisited
        9.6.1 The Object Model of Smalltalk [including aux material on CD and questions]

    10 Functional Languages
    10.1 Historical Origins
    10.2 Functional Programming Concepts
    10.3 A Review/Overview of Scheme
        10.3.1 Bindings
        10.3.2 Lists and Numbers
        10.3.3 Equality Testing
        10.3.4 Control Flow and Assignment
        10.3.5 Programs as Lists
        10.3.6 Extended Example: DFA Simulation

# Optional:

    9.5 Multiple Inheritance [including aux material on CD and questions]
```

Tuesday November 6, 7:00 pm PDT:

```
    9.4 Dynamic Method Binding
        9.4.1 Virtual and Nonvirtual Methods
        9.4.2 Abstract Classes
        9.4.3 Member Lookup
        9.4.4 Polymorphism
        9.4.5 Object Closures
    9.5 Multiple Inheritance [book only]
    9.6 Object-Oriented Programming Revisited
        9.6.1 The Object Model of Smalltalk [including aux material on CD and questions]
    9.7 Summary and Concluding Remarks
```

Tuesday October 30, 7:00 pm PDT:

```
    9.2 Encapsulation and Inheritance
        9.2.1 Modules
        9.2.2 Classes
        9.2.3 Nesting (Inner Classes)
        9.2.4 Type Extensions
        9.2.5 Extending without Inheritance
    9.3 Initialization and Finalization
        9.3.1 Choosing a Constructor
        9.3.2 References and Values
        9.3.3 Execution Order
        9.3.4 Garbage Collection
```

Tuesday October 23, 7:00 pm PDT:

```
    9.1 Object-Oriented Programming
```

Tuesday October 16, 7:00 pm PDT:

```
8.6 Coroutines
    8.6.1 Stack Allocation
    8.6.2 Transfer
    8.6.3 Implementation of Iterators
    8.6.4 Discrete Event Simulation
8.7 Events
    8.7.1 Sequential Handlers
    8.7.2 Thread-Based Handlers
8.8 Summary and Concluding Remarks
```

Tuesday October 9, 7:00 pm PDT:

```
8.4 Generic Subroutines and Modules
    8.4.1 Implementation Options
    8.4.2 Generic Parameter Constraints
    8.4.3 Implicit Instantiation
    8.4.4 Generics in C++, Java, and C#
8.5 Exception Handling
    8.5.1 Defining Exceptions
    8.5.2 Exception Propagation
    8.5.3 Implementation of Exceptions
```

Tuesday October 2, 7:00 pm PDT:

```
8.1 Review of Stack Layout
8.2 Calling Sequences
    8.2.1 Displays
    8.2.2 Case Studies: Con on the MIPS; Pascal on the x86
    8.2.3 Register Windows
    8.2.4 In-Line Expansion
8.3 Parameter Passing
    8.3.1 Parameter Modes
    8.3.2 Call-by-Name
    8.3.3 Special-Purpose Parameters
    8.3.4 Function Returns
```

Tuesday September 25, 7:00 pm PDT:

```
7.7 Pointers and Recursive Types
    7.7.1 Syntax and Operations
    7.7.2 Dangling References
    7.7.3 Garbage Collection
7.8 Lists
7.9 File and Input/Output
7.10 Equality Testing and Assignment
7.11 Summary and Concluding Remarks
```

Tuesday September 18, 7:00 pm PDT:

```
7.3 Records (Structures) and Variants (Unions)
    7.3.1 Syntax and Operations
    7.3.2 Memory Layout and Its Impact
    7.3.3 `With` Statements
    7.3.4 Variant Records (Unions)
7.4 Arrays
    7.4.1 Syntax and Operations
    7.4.2 Dimensions, Bounds and Allocation
    7.4.3 Memory Layout
7.5 Strings
7.6 Sets
```

Tuesday September 11, 7:00 pm PDT:

```
7.1 Type Systems
    7.1.1 Type Checking
    7.1.2 Polymorphism
    7.1.3 The Meaning of "Type"
    7.1.4 Classifications of Types
    7.1.5 Orthogonality
7.2 Type Checking
    7.2.1 Type Equivalence
    7.2.2 Type Compatibility
    7.2.3 Type Inference
    7.2.4 The ML Type System
```

Tuesday September 04, 7:00 pm PDT:

```
6.5 Iteration
    6.5.1 Enumeration-Controlled Loops
    6.5.2 Combination Loops
    6.5.3 Iterators
    6.5.4 Ordering within Expressions
    6.5.5 Generators in Icon
    6.5.6 Logically Controlled Loops
6.6 Recursion
    6.6.1 Iteration and Recursion
    6.6.2 Applicative-and Normal-Order Evaluation
6.7 Nondeterminancy
6.8 Summary and Concluding Remarks
```

Tuesday August 28, 7:00 pm PDT:

```
6.1 Expression Evaluation
    6.1.1 Precedence and Associativity
    6.1.2 Assignments
    6.1.3 Ordering within Expressions
    6.1.4 Short-Circuit Evaluation
6.2 Structured and Unstructured Flow
    6.2.1 Structured Alternatives to goto
    6.2.2 Continuations
6.3 Sequencing
6.4 Selection
    6.4.1 Short-Circuited Conditions
    6.4.2 Case/Switch Statements
```

Tuesday August 21, 7:00 pm PDT:

```
4.1 The Role of the Semantic Analyzer
4.2 Attribute Grammars
4.3 Evaluating Attributes
4.4 Action Routines
4.5 Space Management for Attributes
4.6 Decorating a Syntax Tree
4.7 Summary and Concluding Remarks
```

Tuesday August 14, 7:00 pm PDT:

```
3.5 The Meaning of Names within a Scope
    3.5.1 Aliases
    3.5.2 Overloading
    3.5.3 Polymorphism and Related Concepts
3.6 The Binding of Referencing Environments
    3.6.1 Subroutine Closures
    3.6.2 First-Class Values and Unlimited Extent
    3.6.3 Object Closures
3.7 Macro Expansion
3.8 Separate Compilation
3.9 Summary and Concluding Remarks
```

Tuesday August 7, 7:00 pm PDT:

```
3.1 The Notion of Binding Time
3.2 Object Lifetime and Storage Management
    3.2.1 Static Allocation
    3.2.2 Stack-Based Allocation
    3.2.3 Heap-Based Allocation
    3.2.4 Garbage Collection
3.3 Scope Rules
    3.3.1 Static Scoping
    3.3.2 Nested Subroutines
    3.3.3 Declaration Order
    3.3.4 Modules
    3.3.5 Module Types and Classes
    3.3.6 Dynamic Scoping
3.4 Implementing Scope
```

Tuesday July 31, 7:00 pm PDT:

```
2.3 Parsing
    2.3.1 Recursive Descent
    2.3.2 Table-Driven Top-Down Parsing
    2.3.3 Bottom-Up Parsing
    2.3.4 Syntax Errors
```

Tuesday July 17 Meeting

```
2.1 Specifying Syntax: Regular Expressions and Context-Free Grammars
    2.1.1 Tokens and Regular Expressions
    2.1.2 Context-Free Grammars
    2.1.3 Derivations and Parse Trees
2.2 Scanning
    2.2.1 Generating a Finite Automaton
    2.2.2 Scanner Code
    2.2.3 Table-Driven Scanning
    2.2.4 Lexical Errors
    2.2.5 Pragmas
```

Thursday July 12 Meeting

Had discussions surrounding the **Check your understanding** sections on pages 16, 25, and 35.

Possible Future Books

Managing Gigabytes, by Ian H Witten.

Object-Oriented Programming: An Evolutionary Approach, by Brad J Cox.

The Art of Unix Programming, by Eric Steven Raymond.

Information Retrieval: Implementing and Evaluating Search Engines, by Some people.

Communicating Sequential Processes, by C. A. R. Hoare

Object-Oriented Programming With ANSI-C, Axel Schreiner

Past Books

Programming Languages Pragmatics, by Michael L Scott

Hacks 'n Koans

Apropos of our git reading, here's an interesting and useful git alias. Paste it into your ~/.gitconfig in the [alias] section (man git-config if you've never edited your .gitconfig before):

```
# "blameh" is for doing "git blame" when you want to see information on
# *every* commit on the current branch that affected a given range of lines
# in a file.  There is a plenitude of git-fu-for-aliases in here.
#   Syntax:       git blameh FILE LINESPEC
#   Description:  Show historical culpability for $FILE, limited by $LINESPEC
#
#   For the full syntax of LINESPEC, see git-blame(1) (the "-L" option).
#   Basic $LINESPEC:
#     1234      show lines 1234 to EOF
#     /foo/     show first line matching /foo/ to end (slashes required)
#     1234,1240 show lines 1234 to 1240
#     1234,+3   show 3 lines starting at 1234 (i.e., 1234 to 1236)
#     1234,-2   show 2 lines culminating at 1234 (i.e., 1233 to 1234)
#   Example:
#     # Somebody broke Whoops::borked() a while ago; show who did what when.
#     % git blameh Whoops.pm '/sub borked/,/^}/'
#   Notes:
#     Aliases that shell out (i.e., those whose values begin with "!")
#       are run from the top-level directory of the repo; "cd $GIT_PREFIX"
#       is to put us back in our working directory.
#     "git blame ... 2>/dev/null || git show ..." is because there's no way
#       to tell "git blame" *not* to barf if $LINESPEC isn't valid for the
#       given commit, and that's a common use case for this alias. Plus,
#       *I* would like to see any commits that I'm glossing over.
#     The arcane choices for single quotes, double quotes, and backslashes
#       are intentional.  You can tweak them, but it's not trivial.
blameh      = !sh -c "'\
              cd \"$GIT_PREFIX\"; \
              git rev-list HEAD \"$1\" | while read cmt; do \
                git blame \"-L$0\" $cmt -- $1 2>/dev/null || git blameh-show $cmt; \
                echo \"  ---\"; \
              done'"
# "blameh-show" is broken out as its own alias to show how one alias can
#   call another, and also so that you can easily customize that piece.
#   E.g., you might want to change "--pretty=oneline" to "--pretty=raw", or
#   to 'exit 0' if you don't want to show skipped commits at all.
blameh-show = show --quiet --abbrev-commit --pretty=oneline
```