CMPSCI 630 Systems **Spring 2022**

Lecture 9

Lecturer: Emery Berger *Scribe: Alex Scarlatos, Nigel Fernandez*

9.1 Ontogeny Recapitulates Phylogeny

The biogenetic law, commonly stated as ontogeny recapitulates phylogeny, theorizes that the stages an animal embryo undergoes during development are a chronological replay of that species' past evolutionary forms [1]. Here, ontogeny is the *growth of an individual*, more formally the origination and development of an organism, usually from the time of fertilization of the egg to becoming an adult. Phylogeny is the *growth of a species*, more formally the study of the evolutionary history and relationships among or within groups of organisms.

In biology, Haeckel's biogenetic law has been discredited as a pseudo scientific notion by the results of experimental embryologists in the early twentieth century. However, in the field of systems, ontogeny recapitulating phylogeny seems applicable to an extent, as every new technology seems to recreate the whole evolution of previous technologies. In class, we discussed three areas in systems where this law can be observed, namely 1) microprocessor architectures (Section 9.2), 2) programming languages (Section 9.3), and 3) operating systems (Section 9.4).

9.2 Microprocessor Architectures

Initially, designers thought that complicated features were not needed for microprocessors because they would not be used to do sophisticated things. The first microprocessors when released were quite primitive with the following characteristics:

Feature	Type (start)
Architecture	8-bit
Memory protection	No
Math instructions	No
Size of register set	Small
Microarchitecture	Trivial
Cache	No

However over the years, people start using microprocessors for complex tasks, and they evolved with increasing complexity, replaying the evolution of previous microprocessors, becoming just as complicated as previous existing microarchitectures.

Feature	Type (start)	Type (evolution)
Architecture	8-bit	16/32/64-bit
Memory protection	No	Yes (MMU)
Math instructions	No	Yes
Size of register set	Small	Expanded
Microarchitecture	Trivial	Complicated
Cache	No	Yes[01]

9.3 Programming Languages

The design of a new programming language goes through a similar cycle as microprocessors seen above, with the following initial characteristics:

Feature	Type (start)
Translation	Interpreter (bytecode / AST)
Garbage collector	Reference counting
Optimization	No
Typing	Dynamic
Scoping problems	Yes
Paradigm	Sequential

Take LISP as an example - the first version of LISP was an interpreter (AST-based) with primitive reference counting garbage collection, dynamically typed, accidentally used dynamic scope instead of lexical scope, and was sequential. Perl, Ruby, Python, JavaScript also have similar phenomenon.

Java was originally an interpreted language using interpreted bytecode with a primitive garbage collector. However, the first versions of Java were statically typed with almost no scope problems, serving as small exceptions.

Similar to microprocessors, when the scale of people using programming languages increased to find that most languages were slow, languages evolved with more complex features.

Feature	Type (start)	Type (evolution)
Translation	Interpreter (bytecode / AST)	Compiler/JIT compiler
Garbage collector	Reference counting	Sophisticated
Optimization	No	Yes
Typing	Dynamic	Static[02]
Scoping problems	Yes	No[03]
Paradigm	Sequential	Threads, Async

Exceptions. Some exceptions to the development cycle of usual programming languages exist. Kotlin and Scala are built on top of the JVM. This allowed the languages to not create a new garbage collector, or a new compiler, or new libraries; they were able to build on the infrastructure which already existed and jump past the evolution phase of other languages. Another similar example is Rust, which is built on top of LLVM, a compiler infrastructure.

[01] Deep memory hierarchy with many levels of data cache and instruction cache.

[02] Some dynamic languages now have gradual typing, which allows for static typing on parts (but not all) of a program. Python uses MyPy for this, and JavaScript uses TypeScript.

[03] Scope problems are not completely fixed in JavaScript for backwards compatibility, and Python enjoys having scoping problems.

9.3.1 [Digression] Pass by Copy in ALGOL 60

There was a mistake in semantics in the ALGOL 60 language. Instead of passing parameters by value or by reference, ALGOL 60 instead had pass by copy, which required making a deep copy of every data structure. For example, counting the number of nodes in a graph required a deep copy of the graph. Although this was a mistake, copying everything and throwing away after use (similar to functional languages), does provide very clean semantics.

9.3.2 [Digression] Copy-on-write

To create a new process from an existing process in UNIX, a command *fork* is used, which creates a new process by duplicating the calling process. The only difference between the two processes (original and duplicated) is the return value of the fork call. The return value of the original calling process is 0 while for the duplicated process it's the process ID (PID).

Copy-on-write is a a neat memory management technique to avoid copying the entire process memory during fork, except for write operations. Copy-on-write can be implemented efficiently using page tables where the page table of the duplicated process has pointers to the original memory but with read-only access [1]. If the duplicated process attempts a write operation, then a copy is created in the form of a new page, and the pointer in the page table is modified to point to the new page.

9.4 Operating Systems

The phenomena of ontogeny recapitulates phylogeny repeats in operating systems as well.

Feature	Type (start)	Type (evolution)
No. of users	Single	Multiple
No. of processes	Single	Multiple
Memory protection	No	Yes
Virtual memory	No	Yes
File system	Flat	Hierarchical
File links	No	Symbolic and hard

9.4.1 Development of Multics

Multics was a highly ambitious OS project with influential features ahead of its time. It was a joint project between MIT, GE, and Bell Labs, and was the precursor to Unix.

Hierarchical File System. Multics introduced the hierarchical file system (HFS). A HFS organizes files in a hierarchy of directories and sub-directories, each containing files or recursive sub directories as seen in Figure 9.1. Although natural and intuitive, this was an advancement over the flat name space file system used at the time where all files were at the same directory level. A flat name space doesn't scale well as the number of files increases.

Symbolic and Hard Links. Multics introduced the notion of links, of which there are two types - symbolic links (symlinks) and hard links. Underneath the file system, files are represented by inodes. A file in a file

[1] Read-only access to the original memory for the duplicated process is enforced using hardware page protection which triggers a page fault if write operations are attempted on the original memory pages.

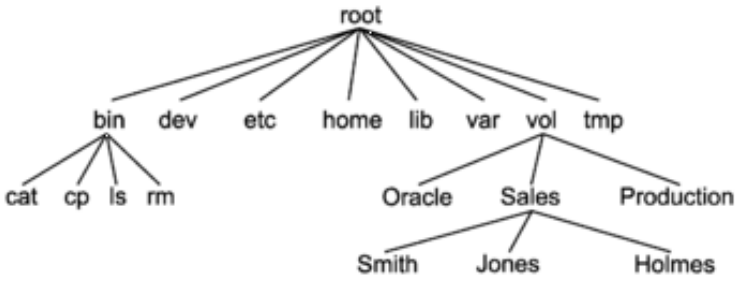

Figure 9.1: An example hierarchical file system structure [3].

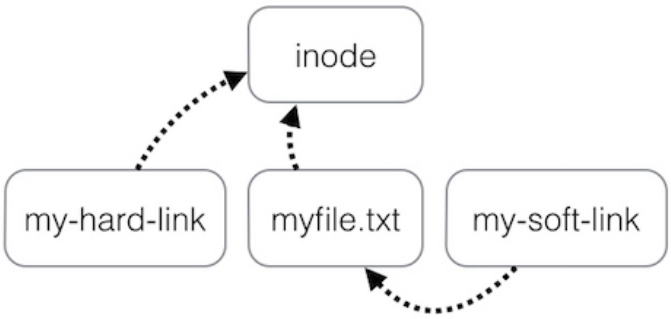

Figure 9.2: Symbolic vs hard links [2]

system is a link to an underlying inode. A hard link creates a new file with a link to the same underlying inode. A symbolic link on the other hand is a link to another file name in the file system as seen in Figure 9.2. Symlinks can be thought of as weak references, where as hard links can be thought of as strong references.

If the original file is renamed or moved, a hard link will be unaffected. Deleting the original file will also not affect a hard link since the underlying inode still exists. However, if the original file is renamed, moved or deleted, a symlink will simply point to a non-existing file [4].

The OS frees (reclaims the memory used by) an inode when all references to it (hard links) are gone. To solve this, file systems, starting with Multics, used reference counting (RC) garbage collection (GC) for files. Creating a hard link to a file increases the reference count by one, and deleting a hard link (or the "original" file) decreases the count by one. When the reference count to the inode goes to zero, the inode can be deleted. As we know, cycles will break RC GC, since an unreachable object can still have pointers to it, preventing the count from going to 0 and thus leaking memory. To avoid cycles, hard links can only be made to files, not directories, allowing RC GC to run without leaks. Preventing hard links to directories for this reason is an example of a leaky abstraction. If a user wants to link to a directory, they can use a symlink, which (as weak pointers) allow RC GC to continue functioning.

Other Features. Multics, unlike its predecessors, was written in a high level language called PL/1 (programming language one), an ALGOL-like language developed by IBM. Multics also introduces a shell, security rings, privileged instructions which allowed virtualisation and virtual machines, worked on multi processors, and had fault tolerance which ensured Multics kept running if one of the processors failed.

Multics is an infamous example of a major slipped software project with development behind schedule,

causing Bell Labs to withdraw from the project. The developers from Bell Labs went on to create Unix, which started more as a toy project for internal use with Bell Labs.

9.4.2 Development of Unix

Two former Bell Lab programmers who had worked on the Multics OS created Unix. A motivation for creating Unix was to run a "video" (text-based) game built for Multics. It was also designed to work on a less powerful machine, the PDP-11. Many of the innovations that follow were driven by the need to create a lightweight and simple-to-implement OS given the small the team.

Name. The name Unix was a play on words - uni (one) vs multi (many). Additionally, Unix is a homonym for eunuchs (castrated men), referencing the fact that Unix was a sort of reduced version of Multics. Yes, this is strange.

Abstraction as Files. Unix was abstraction-heavy, treating almost everything as a file or pseudo file [5]. There were generally 3 kinds - standard files, directories, and "special" files which could act as device interfaces. Some examples of special files are the stdin and stdout streams. File abstraction has turned out to be powerful and ubiquitous, and simplified the OS's implementation at the time. It allowed for a unified I/O API over most objects the OS could deal with. Unix also introduced the notion of a mountable file system. Since resources were limited and distributed over multiple disks, a mountable file system mounted multiple disks to appear as a cohesive file system, providing a unified view and a scalable file system.

Internally, the contents of a standard file are a list of references to data (data was stored in uniform-sized blocks). If a pointer in the file content points directly to a block, it is called direct. A pointer may also point to a block of pointers to blocks, or may even have a third layer of indirection. These are called indirect. This structure allowed files to be as big as they needed to be, and made data access and file deletion quicker (the alternate structure is a linked list of file blocks which would be traversed in linear time, whereas the Unix tree-like indirection structure can be traversed in log time).

Internally, a directory is a file whose contents references the files within that directory. On the other hand, Multics represented directories using tree data structured like a B-tree.

Buffers. To avoid frequent disc accesses, when reading a file in Unix, the associated block data would be placed in a read buffer in memory, which is where the actual reads would come from. Similarly, when writing to a file, updates were saved in a buffer before being committed to disc. Today, there are many layers of buffers in between each level of abstraction in a computer system - between apps and the OS, between the OS and the file system, between the kernel and the hardware, and then there are even hardware buffers between the different levels in the hardware.

[**Related to buffers**] **Apple** devices today store hard memory writes in a buffer, rather than committing them to hard memory immediately, which saves time. They can do this because most modern Apple devices have a battery, so there is no practical concern of losing the information to a power loss while it sits in the buffer.

[**Aside**] **Emery is from Florida**, apparently the "lightning capital of the world", where once his computer was fried by a power surge through a telephone line to his modem.

Processes. Unix ran each program as a process. A process has its own globals, heap, file handles (references to open files), translation lookaside buffer (TLB, a cache for virtual to physical memory translation), etc. Threads, on the other hand, share all of this information, except each thread has its own stack.

Composition Commands. Composition commands can be formed using pipes such as `ls | wc -l`, which sends the stdout of the first program to the stdin of the second program. This allowed increased functionality

by combining simpler functions, and avoided having to implement an extensive list of features for each program, as was the strategy for Multics. Another example of composition is `cut -f 1 -d , | sort | uniq -c` which takes a csv as an input, extracts the first field where there is a comma, combines and sorts all of them to produce a histogram. Composition gave rise to mini programming languages like AWK and sed which popularized regular expressions. Perl was initially parsed by using this workflow, but as system calls are expensive, eventually moved the implementation to its interpreter.

Other Features. Unix also featured multiple users and multiple processes running at the same time, but did not have transactions at the time of implementation.

9.4.3 Other Examples of Ontogeny Recapitulates Phylogeny

DOS. CP/M and MS-DOS originally started with a very stripped down set of features, despite being created in the post-Unix era:

Feature	Type (start)
No. of users	Single
No. of processes	Single
Memory protection	No
Virtual memory	No
File system	Flat
File links	No

Today, of course, we have Windows, which is a real operating system with all the features we know and love (especially the lack of IE).

Apple computers followed the same path, and mobile operating systems did as well. Another example is embedded systems.

9.4.4 [Digression] CAP Theorem

The CAP theorem proposed by Eric Brewer states that in a distributed system you can have two of the following three guarantees, namely 1) consistency, 2) availability, and 3) partition tolerance. Emery didn't seem to think the theorem needed a proof and that it was a rather obvious property, so we probably don't need to worry about it much.

9.5 Why is ontogeny recapitulates phylogeny applicable in systems?

Why, in programming languages for instance, does every new language repeat the same cycle of evolution, reinventing the wheel to a large extent? Here are some theories.

Inevitability of novelty. The whole point of developing a new language is to do stuff sufficiently differently from existing languages. If we could just copy and paste the code from existing implementations, then there would be no point in making a new language. Thus, every advancement in PL must be reinvented to a small extent. This theory is okay, but the Kotlin/Scala/Rust exceptions serve as counterarguments, since they achieved novelty without having to re-implement every feature.

Worse is better. Development of a new languages needs rapid prototyping with a focus to ship fast, rather than solving all possible problems perfectly and risk being obsolete by the time all the kinks are worked out. "Worse is better" advocates new programming languages to follow the same evolutionary pattern of starting with simple features and shipping soon, and later evolving to add complex features when there is demand for it by the community. This seemed to be the consensus theory in class.

They're all fools! I can't believe the guy who built JavaScript in 2 weeks was such an idiot! Didn't he know what static typing was? And now we're stuck with *var* forever! *End obvious sarcasm*. The argument here is that either 1) the people who invented these languages had no idea what they were doing, or 2) inventions, such as garbage collection and lexical scoping, are poorly documented by the community. This is not a great argument, since the work done by the inventors of these languages proves that they do indeed know what they're doing, and that the pared down implementations were done on purpose. I think the bad documentation argument could also be disproven by the fact that so many languages (and architectures, etc.) end up using the same concepts, unless we assume convergent evolution in all these cases, which is obviously not the case.

References

[1] Ernst haeckel's biogenetic law (1866). `https://embryo.asu.edu/pages/ernst-haeckels-biogenetic-law-1866`. Accessed: 2022-03-18.

[2] Hard links and symbolic links — a comparison. `https://medium.com/@307/hard-links-and-symbolic-links-a-comparison-7f2b56864cdd`. Accessed: 2022-03-18.

[3] Unix directory structure. `https://www.tau.ac.il/~tsirel/dump/Static/knowino.org/wiki/Unix_directory_structure.html`. Accessed: 2022-03-18.

[4] What is the difference between a symbolic link and a hard link? `https://stackoverflow.com/questions/185899/what-is-the-difference-between-a-symbolic-link-and-a-hard-link`. Accessed: 2022-03-18.

[5] Dennis M Ritchie. The evolution of the unix time-sharing system. In *Symposium on Language Design and Programming Methodology*, pages 25–35. Springer, 1979.