Open Source Software Development

Introduction to
Open Source Software (OSS)

Thái Minh Tuấn - Email: minhtuan@ctu.edu.vn

Learning Objectives

- What is Free and Open-Source Software (FOSS)?
- Proprietary (Closed Source) Software
- OSS philosophical strains: Pragmatism vs. Idealism
- History of Open Source Software
- Reasons for Using Open Source Software
- Successful OSS Projects

What is Free and Open-Source Software (FOSS)?

- Free as in freedom to do something
  - NOT as in no cost
- Source code is made available
- With a license which provides rights (4 essential freedoms):
  - To run the program as you wish, for any purpose (freedom 0)
  - To study how the program works, and change it (freedom 1)
  - To redistribute copies (freedom 2)
  - To distribute copies of your modified versions (freedom 3)
  - Without restriction on the user's identity or purpose
- There is a multiplicity of licensing methods, falling into the two general classifications:
  - Restrictive (e.g., GPL-licensed software) vs. Permissive (e.g., BSD-licensed software)
  - (will discuss later)

Proprietary (Closed Source) Software

- Only the owners have full legal access to the source code
  - May grant inspection rights to trusted partners with non-disclosure agreements (NDAs).
- The owners may or may not be the authors of the code
  - The authors have been working under contracts giving up their individual rights
  - Many proprietary softwares are no longer even owned by their originators
    - Passed on through sales over time
- End users must accept a license restricting their rights
  - Through a confusing click-through box presented after a very long and dense description
- Price is not the point:
  - One can charge as much as one wants for an open product
  - One can provide a proprietary product for free
- The license differences have to do with redistribution, modification, reuse of code, etc.

OSS philosophical strains: Pragmatism vs. Idealism

- Idealism: There is a profound belief that all softwares should be open for ideological and ethical reasons, not just technological ones.
- Pragmatism: The primary considerations are technical ones, including faster and better development involving more contributors and review, easier debugging, etc.
- Ideological perspective also has strong technical imperatives, and in many cases, the objectives of both streams coincide
  - Should the software powering a life-saving medical device (such as a pacemaker, or an insulin pump) be secret?
  - Should the software powering voting machines be closed

History of Open Source Software (1/4)

- Open Source Software (OSS) has a long history, although the actual term only dates back to 1998:
- 1950s:
  - Software arose from researchers, both academic and corporate
  - Distributed openly and cooperatively
  - Source always distributed, binaries less often
  - Software not seen as separate commodity
  - Software bundled for free with hardware
  - Licensing was sloppy
- 1960s:
  - Major aspects of computer science and software rapidly developed both in academia (MIT, UC Berkeley) and industrial research labs (Bell Labs, Xerox)
  - 1968: ARPANET emerges, eventually leads to Internet, essential for developers and researchers to communicate, share, collaborate
  - 1969: UNIX given birth at Bell Labs (AT&T) and given for free to universities and research centers
  - 1969: IBM forced by US government to break software and hardware apart and sell/distribute separately, due to unfair business practices.

History of Open Source Software (2/4)

- 1970s:
  - 1976: emacs released by Richard Stallman and Guy Steele. Versions by other authors proliferate, including proprietary ones. GNU emacs not released until 1985
  - 1978: First version of TeX released by Donald Knuth, an open source typesetting system often used for publishing journal articles and books, still in widespread use, usually in the LaTeX version.
- 1980s:
  - 1980: Usenet begins as the ancestor of user forums and the World Wide Web
  - 1982: GNU project announced by Richard Stallman
  - 1984: X Window System released out of MIT, with X11 protocol released in 1987 (now run by X.Org)
  - 1985: Free Software Foundation (FSF) founded by Richard Stallman
  - 1987: gcc released (now known as Gnu Compiler Collection)
  - 1987: Perl released by Larry Wall.

History of Open Source Software (3/4)

- 1990s:
  - 1991: Linux begun by Linus Torvalds
  - 1992: Python released by Guido Van Rossum
  - 1992: 386BSD released
  - 1992: Samba developed by Andrew Tridgell in Australia
  - 1993: Debian first released by Ian Murdock, and still survives as the largest non-commercial Linux distribution
  - 1993: Red Hat is founded; while earlier commercial Linux distributions already existed, Red Hat was the first company built on open source to become very large
  - 1993: NetBSD released as a fork from 386BSD
  - 1993: FreeBSD released
  - 1993: Wine released to run Windows applications on Linux
  - 1994: MySQL development begins in Sweden, first release in 1995
  - 1995: PHP, GIMP, and Ruby released
  - 1996: Apache web server released
  - 1996: KDE released
  - 1997: GNOME released
  - 1998: Netscape open sources its browser, which will later become Firefox
  - 1999: OpenOffice released (eventually forks into LibreOffice).

History of Open Source Software (4/4)

- 2000s:
  - 2000: LLVM compiler project begun at UI-Champaign
  - 2002: Blender released as an open source project
  - 2003: Firefox released
  - 2004: Ubuntu releases its first version, which Canonical builds on top of Debian
  - 2005: Git released by Linus Torvalds
  - 2007: Android released, based on Linux kernel; first devices on the market in 2008
  - 2008: Chromium released by Google; basis of Google Chrome.
- 2010s:
  - In this decade, OSS becomes ubiquitous, dominating the smartphone market, supercomputing, worldwide networking infrastructure, etc.

Open Source Governance Models - Company-Led

- Company-Led (mostly, a closed process): Google Android, Red Hat Linux
  - Development is strongly led by one corporate or organizational interest
  - One entity controls software design and releases
  - May or may not solicit contributions, patches, suggestions, etc.
  - Internal discussions and controversies may not be aired very much
  - It is not definitively known what will be in the next software release
  - Upon release, all software is completely in the open

Open Source Governance Models - Benevolent Dictatorship

- Benevolent Dictatorship (strong leadership): Linux kernel
  - One individual has overriding influence in every decision
  - Project's quality depends on that of the dictator
  - Can avoid endless discussions and lead to quicker pace
  - As project grows, success depends critically on the dictator's ability to:
    - Handle many contributors
    - Use a sane, scalable version control system
    - Appoint and work with subsystem maintainers
  - The dictator's role may be social and political, not structural (forks can occur at any time)
  - Maintainers write less and less code as projects mature

Open Source Governance Models - Governing Board

- Governing Board (tighter control by smaller groups): FreeBSD, Debian
  - A body (group) carries out discussions on open mailing lists
  - Decisions about design and release dates are made collectively
  - Decisions about who can contribute, and how patches and new software are accepted, are made by the governing body
  - There is much variation in governing structures, rules of organization, degree of consensus required, etc.
  - Tends to release less frequent, but hopefully well-debugged versions

Exercise: Basic Facts about Open Source Software

- Which of the following statements are true?
  - 1. Once fees are charged for for the use or distribution of software, it can no longer be termed open source.
  - 2. If the control, content and timing of the official software releases is controlled by one entity (such as a company), it can no longer be termed open source.
  - 3. The first widely used OSS product was the Linux kernel, first released in 1991.
  - 4. If a program is written using an open source computer language (such as Python or Perl) it must be released under an open source license.

Reasons for Using Open Source Software - Collaborative Development

- Collaborative Development: Enables software projects to build better software
  - NOT everyone has to solve the same problems and make the same mistakes
  - Softwares can be made much faster and costs can be reduced
  - Stronger and more secure code
    - More eyeballs viewing code and more groups testing
  - It is often hard for competitors to get used to the idea of sharing
    - Competitors can compete on user-facing interfaces
    - End users still see plenty of product differentiation and have varying experiences

Reasons for Using Open Source Software - Security and Quality of Source Code:

- Code published openly tends to be cleaner
  - It is embarrassing to show ugly, sloppy code
  - Coding standards and styles tend to be cleaner and more consistent on community projects
  - More people have to understand and work on the code
- More eyeballs examining code and looking for security weaknesses
  - Before they are discovered by bad actors
- There is more input in original design to avoid bad ideas
- No *security through obscurity*
- No *just trust me*
- Potentially faster bug repair

Reasons for Using Open Source Software - Advantages for Various Stakeholders (1/3)

- Individual users:
  - Can mix and match software from different sources
  - Can save money on buying and leasing software
  - Can avoid vendor lock-in, maintaining choice
  - Can look under the hook ("trust, but verify")
  - More funs
- Business:
  - Collaborative development
    - Lowers total cost of development
    - Speeds up time to market
    - Work is submitted to wider community for criticism, suggestions and contributions
    - Uses well-delineated application programming interface (API)
  - Marketing
    - Customers know what they are getting - They have confidence in quality, there are no secrets
    - Product is seen as part a large ecosystem of related products
    - More flexible, possibly modular construction
    - Adoption by larger community can help build customer's confidence about product's durability and stability

Reasons for Using Open Source Software - Advantages for Various Stakeholders (2/3)

- Education - Elementary, High school and Public systems:
  - Very large amount of available teaching resources at little and no cost
  - Very wide range of areas available for using, operating and system administration, and programming
  - Students do not become locked into vendor products
  - Generally lower hardware cost, and easier to use old hardware
  - Student are learning the skills they will need in the workforce
  - Unleashes student creativity: more fun!
- Education - University:
  - Students can study and work on the internal of operating systems, applications and libraries, system administrator utilities
  - Student are ready to enter to the workforce where they are most needed
  - Good habits are developed, including how to work with the open source community
  - Student work is easy for prospective employers to evaluate, since it is publicly accessible

Reasons for Using Open Source Software - Advantages for Various Stakeholders (3/3)

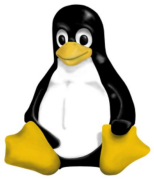
- Developer
  - No need to reinvent everything
  - Help to make good, early decisions on product design
  - More eyeballs on code can fix bugs faster
  - Suggestions and contributions are provided by a large group of developers
  - Great for finding next job
    - Code is readily available for evaluation
    - Can demonstrate how well you work and play with others
    - Can show how good you are at mentoring and maintaining projects and subprojects
    - Know you are not alone

OSS - Fear, Uncertainty and Doubt (FUD)

- Misinformation to influence recipients to avoid certain strategies, products or classes of products by appealing to fear
- Microsoft was widely accused of spreading FUD about Linux in the 1990s.
  - In present day, Microsoft has stopped doing so and is actually employing OSS widely
- OSS - FUD statements
  - OSS is a virus. If you include it in your product, all your source must be made available to everyone
  - OSS infringes on software patents, and the related claim that it forces you to grant patent rights to others
  - OSS products leave nowhere to turn when they break, or to get technical help
  - OSS requires a lot of legal help to avoid the above pitfalls, and is thus very expensive

Successful OSS Projects - Linux Kernel

- Has been an open source project since its inception in 1991
- Basis of almost all of the world's computing infrastructure
  - From the most powerful supercomputers to the largest number of mobile devices
- Become ubiquitous in an enormous range of devices and embedded products:
  - Personal fitness devices , DVR boxes and televisions, automotive systems, In-flight entertainment systems, medical devices, etc.
- Development community is large and mature; headed by Linus Torvalds
  - Well-formed organizational structure of subsystem maintainers and mailing lists, contributor guidelines and methods
  - Thousands of developers
    - From hardware and software companies; industrial collaborative consortia
- Released under GPL Version 2
  - Some sections that have dual licensing, with other open source licenses
  - New version is put out every 10-12 weeks



Successful OSS Projects - Git

- A distributed version control system
  - Used worldwide for an astounding number of collaborative products
- Created by Linus Torvalds in 2005
  - To handle the increasingly difficult task of coordinating and consolidating the work of thousands of contributors to the Linux kernel
- The basis of GitHub (https://github.com)
  - Hosts more than one hundred million of open source projects repositories
- Can be used in non-open source projects (like many other tools)
  - No way contaminates using it for OSS work

Successful OSS Projects - Apache

- Work on the Apache HTTP Server began in 1995.
  - The most widely used web server, with roughly 50% of the market share
- Operates under the umbrella of the Apache Software Foundation (ASF)
- Released under the Apache Software License
  - Has also been adopted by many other projects.
  - It is more permissive than the GPL

Successful OSS Projects - Python, Perl and Other Computer Languages

- Many computing languages are developed using open source methods.
  - Python is an interpreted, high-level, general-purpose programming language
    - The most popular programming language[1]
  - Perl is a programming language specially designed for text editing
    - Now widely used for a variety of purposes including Linux system administration, network programming, web development, etc.
  - Many others such as Ruby, GCC, and Rust

1. https://statisticstimes.com/tech/top-computer-languages.php

Successful OSS Projects - GNU: gcc, gdb, and More

- GNU (Gnu's Not Unix) Project has provided many essential ingredients for virtually all modern computer technologies,
  - Under various versions of the GPL (General Public License)
  - Started in 1983 by Richard Stallman
- Some of the most prominent products emanating from the GNU umbrella
  - GNU Compiler Collection (gcc): Compiler system supports various programming languages
  - GNU Debugger (gdb): Portable debugger that runs on many Unix-like systems
  - GNU C Library (glibc): GNU Project's implementation of the C standard library
  - GNU Bash (bash): Unix shell and command language used widely as the default login shell for most Linux distributions
  - GNU Core Utilities (coreutils): A package containing re-implementations for many of the basic tools, such as cat, ls, and rm, which are used on Unix-like operating systems

Successful OSS Projects - X and Desktop Managers

- Used to instantiate the GUI seen on any Linux laptop or workstation
- X Window System: The underlying software that handles basic screen, input device and other point operations
  - A newer and more secure alternative, Wayland, is moving into its current role
- GNOME, KDE, XFCE: Desktop Manager frameworks that control the operation of graphical interfaces, drag and drop between them, appearance of the desktop

Successful OSS Projects - OpenStack, Kubernetes and Other OSS Projects

- Many other large scale (as well as small) collaborative projects that are based on open source software

Open Source Software Development

OSS Licensing & Legal Issues

Thái Minh Tuấn - Email: minhtuan@ctu.edu.vn

Learning Objectives

- What Are (Software) Licenses?
- Software License Types
- Open Source Definition
- OSS Restrictive (Copyleft/Protective) licenses
- OSS Permissive (Non-protective) licenses
- Software Patents
- Which License to Choose?
- Mix and Match License?
- What Is a Copyright?

What Are (Software) Licenses?

- *"A software license is a legal instrument . . . governing the use or redistribution of software"* (according to Wikipedia).
- Software:
  - A document that represents a program or firmware written by author(s)
  - Software program: A machine executable version of an algorithm or idea
- Legal instrument:
  - Formally expresses a legally enforceable act, process, or contractual duty, obligation, or right, and therefore evidences that act, process, or agreement
- Governing the use:
  - How a person or company can use this software program
  - Publicly post it, use inside their company, or personal use at home?
- Redistribution:
  - How it can be shared
  - Give a copy to a friend, modify or create derivative works, post online, etc.?

Software License Types

- Two broad categories: Proprietary or Open source
- Proprietary licenses:
  - Specific to a company or a project
  - Copyright holder keeps many or most of the rights to themselves
  - Often adds additional restrictions on what users can do
  - Commercial or freely available
- Open-source licenses:
  - Follow the definition of OSS developed by the Open Source Initiative
  - Allow software to be freely used, modified, and shared
  - Permissive or Copyleft
  - Patent Grant or No Patent Grant

Open Source Definition

- For code to be considered open source:
  - The license the code is released under should comply with the properties listed in the definition maintained by the Open Source Initiative (OSI)
- OSI definition
  - 1. Free Redistribution
  - 2. Source Code
  - 3. Derived Works
  - 4. Integrity of The Author's Source Code
  - 5. No Discrimination Against Persons or Groups
  - 6. No Discrimination Against Fields of Endeavor
  - 7. Distribution of License
  - 8. License Must Not Be Specific to a Product
  - 9. License Must Not Restrict Other Software
  - 10. License Must Be Technology-Neutral

OSS Restrictive (Copyleft/Protective) licenses

- Have requirements:
  - How the software can be redistributed
  - Impact how derivative works can be distributed
- The term copyleft originated from the GNU licenses
  - Has been expanded over time to cover other licenses
- Get grouped into strong and weak copyleft licenses

OSS Strong-Copyleft licenses

- Offering people:
  - The right to freely distribute copies and modified versions of a work
    - Provides 4 essential freedoms
  - With the stipulation that the same rights be preserved in derivative works down the line
    - Derivative works, including forks, would need to retain the same license as the original code
      - If you are not the copyright holder of the original code
- Examples:
  - GNU General Public Licenses (GPL-2.0 and GPL-3.0)
  - Affero General Public License (AGPL-3.0)

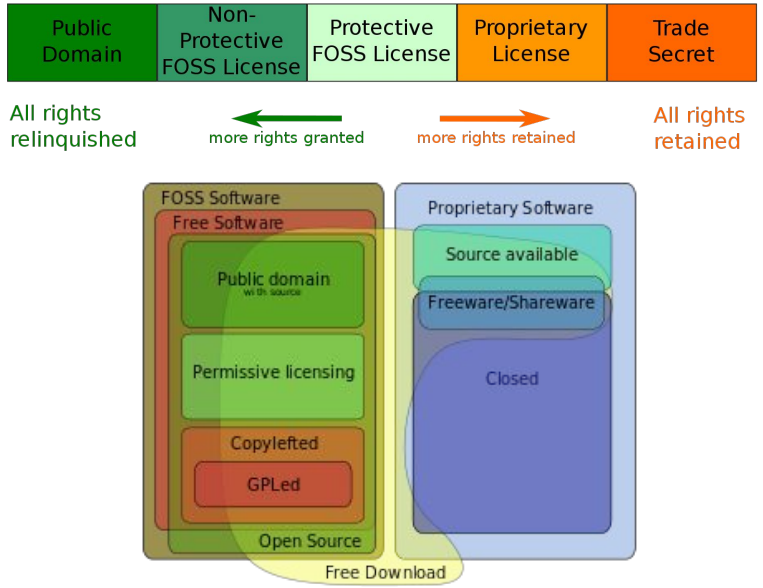
OSS Weak-Copyleft licenses

- Not all derived works inherit the copyleft license
- Generally used for the creation of software libraries
  - Allow other software to link to
  - No legal requirement for being distributed under the library's copyleft license
- Examples:
  - Lesser GNU Public License (written as LGPL-2.0, LGPL-2.1 or LGPL-3.0)
  - Mozilla Public Licenses (written as MPL-1.0, MPL-1.1 or MPL-2.0)
  - Eclipse Public License (written as EPL-1.0 or EPL-2.0)
  - Common Development and Distribution License (written as CDDL-1.0 or CDDL-1.1)

OSS Permissive (Non-protective) licenses

- Minimal requirements on what you must do when redistributing the software
  - Typically limited to things like retaining or delivering attribution notices
- Often easier to be combined with copyleft and/or proprietary software
  - But there still are compatibility issues in some cases
- Common examples:
  - MIT
  - BSD-2-Clause and BSD-3-Clause
  - Apache-2.0

Software licenses



Patents

- A mechanism for protecting intellectual property
  - Existed in various forms since the Middle Ages
- Refers to the right granted to anyone
  - Invents any new, useful, and non-obvious process, machine, article of manufacture, or composition of matter
    - Provides the right to exclude others from making, using, selling, offering for sale, or importing the patented invention
    - For the term of the patent
- Must be filed for in each nation (or trading block such as the European Union)
  - Expensive and time-consuming
- Search for a US patent
- Google Patents

Patents - Example

- Michael Jackson Smooth Criminal Anti Gravity
  - https://www.youtube.com/watch?v=2mhnL1YJ2fA
- Method and means for creating anti-gravity illusion
  - US5255452A
  - Inventor: Michael J. Jackson, Michael L. Bush, Dennis Tompkins
  - https://patents.google.com/patent/US5255452A/en

(Software) Patents

- Gives exclusionary rights to material such as a computer program, library, interfaces or even techniques and algorithms
  - Earliest software patents appear to have been granted in early 1960
- Have been used defensively, with corporations cross-licensing each other's work to avoid litigation
  - However, there are many well-known cases of expensive legal battles
  - Existing copyright and trademark laws are INsufficient for protecting intellectual property
  - Developers and organizations have to learn to deal with them properly
- Software Patent vs. Copyright
  - Copyright only protects the "exact" source code
    - Avoid infringement by making changes to the code
  - Patents protect the novel features, processes, and design of a system that powers a "software/app" rather than just the underlying source code

Express Patent Grant with License? (1/2)

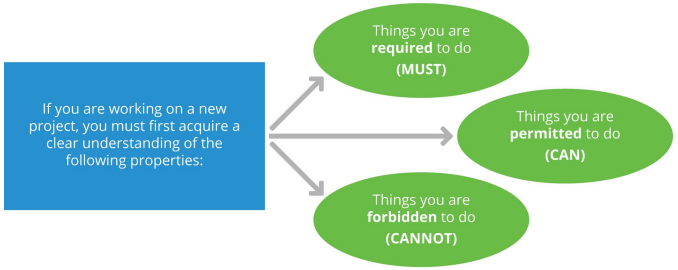
- Software can implement patented inventions
  - Owe the company that patented them a fee to do so
- Some licenses are EXPLICITLY GRANTED the right to use any of the patents the contributors have on methods they have included in the code
  - With no additional charge
- Some licenses also EXPLICITLY EXCLUDE a patent license grant
- Some licenses that does not explicitly reference patents, but then add a grant of patents in a separate file
  - LICENSE file (BSD-3-Clause)
  - Additional PATENTS file of Golang repo on GitHub

Express Patent Grant with License? (2/2)

Explicitly EXCLUDES Patent License	Implicit Patent License AND/OR Patents Not Explicitly Mentioned	Explicitly INCLUDES Patent License
• CC0-1.0 • Other Creative Commons licenses, e.g. CC-BY-4.0 • BSD-3-Clause-Clear	• BSD-2-Clause and BSD-3-Clause • MIT • ISC • Apache-1.0 and Apache-1.1 • GPL-2.0 and LGPL-2.1	• Apache-2.0 • GPL-3.0 and LGPL-3.0 • MPL-1.0, MPL-1.1 and MPL-2.0 • EPL-1.0 and EPL-2.0 • CDDL-1.0 and CDDL-1.1

Which License to Choose?

- License selected is your way of specifying:



- Choosing a standard and commonly-used open source license
  - Help to make it easier for everyone else to understand what their rights and obligations are

Properties to Consider

- It is important to be clear on your goals for releasing the code
  - Who (what types of people/organizations) do you want to adopt it?
  - Do you want to see any changes people make to your code when they redistribute it?
  - Do you want other people to be able to sell your code for a profit?
- Common properties:

• Publish license, copyright notices, change summaries? • Disclose source? • Distribution of modified work? • Sublicensing? • Private or commercial use?	• Patent grant? • Able to use trademarks? • Can code be warrantied? • Able to hold liable for damages? • Scope of license: work as a whole or only specific file?

Open Source License Comparison

- Commercial Use: Software and derivatives may be used for commercial purposes
- Distribution: May distribute this software
- Modification: May be modified
- Patent Use: Provides an express grant of patent rights from the contributor to the recipient. This license explicitly states that it does NOT grant you any rights in the patents of contributors
- Private Use: You may use and modify the software without distributing it
- Disclose Source: Source code must be made available when distributing the software
- License and Copyright Notice: Include a copy of the license and copyright notice with the code

Open Source License Comparison

- Network Use is Distribution: Users who interact with the software via network are given the right to receive a copy of the corresponding source code
- Same License: Modifications must be released under the same license when distributing the software. In some cases a similar or related license may be used
- State Changes: Indicate changes made to the code
- Hold Liable: Software is provided without warranty and the software author/license owner cannot be held liable for damages
- Trademark Use: This license explicitly states that it does NOT grant you trademark rights, even though licenses without such a statement probably do not grant you any implicit trademark rights

Open Source License Comparison Grid

Is There Help Available?(1/2)

- *Open Source Licenses by Category* from the Open Source Initiative lists the approved open source licenses
- *Choose an Open Source License* is sponsored by GitHub.
  - Walking you through the properties you must consider, helping you decide what license makes sense
    - Blue - Rules you must follow
    - Green - Rules you can follow
    - Red - Things you cannot do
- *Various Licenses and Comments About Them* provides a description of many licenses and comments about them
- Creative Commons Licenses help you understand license options for images and documentation

Is There Help Available?(2/2)

- The OSSWatch licence differentiator tool attempts to help its users understand their own preferences in relation to free and open source software licences
- There are 7 choices that need to make
  - 1. Popular and widely used
  - 2. Licence type
  - 3. Jurisdiction
  - 4.a Grants patent rights
  - 4.b Patent retaliation clause
  - 5. Specifies enhanced attribution
  - 6. Addresses ASP/privacy loophole
  - 7. Includes 'no promotion' feature

Mix and Match License?

- When can you combine licenses in a project? There is not a single global answer to this question
  - Interpretations of license compatibility
  - Which licenses are acceptable for inclusion, depend on the project and distribution the project ships in
  - Not all projects treat combinations of licenses the same way
- Guidance created by some of the common projects
  - GNU/Free Software Foundation
  - Fedora Project
  - Debian
  - Android Open Source Project
  - The Apache Software Foundation

REMEMBER:
You should always check with your legal counsel before contributing to a project on behalf of your employer

What Does a Reference to a License Look Like in a File?

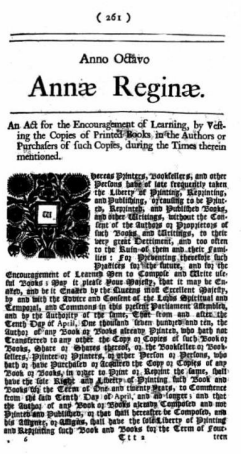
- Full license text is frequently long and complex,
  - NOT always suitable to put in each file in a project
- Common practice:
  - Put a LICENSE file at the top level of the project,
    - Include the full text of the license(s) that may be in effect for the project as a whole
  - For the other files in a project,
    - Referring to the actual license within each file — even if that reference doesn't include the full license text
    - Ambiguity is removed as to which license governs that file,
      - Particularly when there are multiple licenses being used by a project
    - A license reference actually look like in a file
      - A standard header for the license (if it exists)
      - An unambiguous reference to the license intended

What Is a Copyright? (1/2)

- "*Copyright is a legal right created by the law of a country that grants the creator of an original work exclusive rights for its use and distribution*." (according to Wikipedia)
  - Granting to the copyright owner for protection of their work
  - Original "work" can refer to creative, intellectual or artistic output.
    - Software source code
  - Copyright does not cover the ideas or information, but rather how they are expressed (or written down)
    - Ideas and algorithms can be covered by patent law instead
  - The work must be considered as original
- Countries have their own copyright laws
- International treaties (e.g. Berne Convention and the Universal Copyright Convention) have standardized the treatment
  - Local variations in law do exist
  - United States did not sign the Berne Convention until 1989
    - Over 100 years after the Berne Convention was created (1886)

What Is a Copyright? (2/2)

- Great Britain was the first country to recognize the rights of authors
  - The Copyright Act of 1709, or Statute of Anne, was the first law that recognized the rights of authors
  - The law has been reworked and refined over time as technologies changed
- Copyright protection does have limits. In the United States, it does not extend to:
  - Any idea, procedure, process, system, principle or discovery
  - Facts such as names, titles, short phrases, slogans, familiar symbols, typographic variations, lettering, coloring and listing of contents or ingredients


(261)

Anno Octavo

Annæ Reginæ.

An Act for the Encouragement of Learning, by vesting the Copies of Printed Books in the Authors or Purchasers of such Copies, during the Times therein mentioned.

Whereas Printers, Booksellers, and other Persons have of late frequently taken the Liberty of Printing, Reprinting, and Publishing, or causing to be Printed, Reprinted, and Published Books, and other Writings, without the Consent of the Authors or Proprietors of such Books, and Writings, to their very great Detriment, and too often to the Ruin of them and their Families: For Preventing therefore such Practices for the future, and for the Encouragement of Learned Men to Compose and Write useful Books: May it please Your Majesty, that it may be Enacted, and be it Enacted by the Queen's most Excellent Majesty, by and with the Advice and Consent of the Lords Spiritual and Temporal, and Commons in this present Parliament Assembled, and by the Authority of the same, That from and after the Tenth Day of April, One thousand seven hundred and ten, the Author of any Book or Books already Printed, who hath not Transferred to any other the Copy or Copies of such Book or Books, Share or Shares thereof, or the Booksellers or Booksellers, Printer or Printers, or other Person or Persons, who hath or have Purchased or Acquired the Copy or Copies of any Book or Books, in order to Print or Reprint the same, shall have the sole Right and Liberty of Printing such Book and Books for the Term of One and twenty Years, to Commence from the said Tenth Day of April, and no longer: and that the Author of any Book or Books already Composed and not Printed and Published, or that shall hereafter be Composed, and his Assignee or Assigns, shall have the sole Liberty of Printing and Reprinting such Book and Books for the Term of Four-teen

Who Is a Copyright Holder?

- The author of the work is generally considered to be the copyright holder
- When a work was created in the course of the author's employment?
  - It is considered as a "work for hire" and the employer is the copyright holder
- Rights of the copyright holder can also be assigned to another organization or individual
- Some jurisdictions around the world may have different positions on who owns a copyright,
  - Should always check with local counsel first

Copyright Notices

- Under the Berne Convention, an author doesn't need to register their copyright in order to own it
  - The copyright exists upon the creation of the work
- Still a good idea to insert a copyright notice in the body of the work
  - Make it clear that the work is copyrighted
  - Provide guidance about who the copyright holders are

Historical Format:
Copyright (C) 2001, 2004-2006 Company ABC.

Copyright 1998, Linus Torvalds
© 2003, 2010, Free Software Foundation, Inc.
Copyright (C) 2011, 2014-2019 Company ABC.

Newer Format
Copyright Contributors to the _____ Project.
Copyright The _____ Contributors.

Copyright The Kubernetes Authors.
Copyright Contributors to the OpenVDB Project.
Copyright the Hyperledger Fabric contributors.

What Are Valid File Notices?

- Valid file notices have the following characteristics:
  - Define in a clear manner the license that the file is governed by
  - May include information or references to copyright holders
  - Also provide additional information about authorship, when the author is not the copyright holder
- Examples:
  - Linux
  - FOSSology
  - gcc
  - Nova
  - Das U-Boot
  - Zephyr

Open Source Software Development

Working on OSS Projects

Thái Minh Tuấn - Email: minhtuan@ctu.edu.vn

Learning Objectives

- Contributing to OSS Projects
- Respecting and Encouraging Diversity in OSS
- Constructing an Open Source Strategy
- TODO Group
- OpenChain Project, FOSSology, SPDX, CHAOSS
- Leadership vs. Control: Why Do Many OSS Projects Fail?

How to Contribute OSS Projects Properly (1/2)

- Before making contributions to any OSS project you should:
  - Investigate it, understand its workflow and style
  - Identify the scope and nature of your work
- Identify how the project communicates
  - Mailing list, existing archives
  - Joint Internet Relay Chat (IRC) network (if any)
- Understand how contributions are submitted
  - Mailing lists? Email?
  - A revision control system, such as git or subversion?
- Study previous history
  - Has your idea been considered before and rejected?
  - Is there already an individual or group working on the idea?
    - If so, them in many cases, not start over

How to Contribute OSS Projects Properly (2/2)

- Better to offer your services for testing, finding bugs, and similar work
  - Rather than beginning by submitting code
- Be competent at whatever programming or scripting language the project uses
  - Developers will be impatient in correcting your code
- Balance between asking for suggestions and review early in the process
- Be polite, respectful, avoid obscenities, flaming and trolling

Contributing to OSS Projects

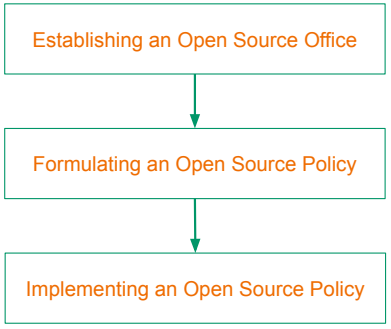
- Things to consider when contributing to OSS projects:
  - Study and Understand the Project DNA
  - Figure Out What Itch You Want to Scratch
  - Identify Maintainers and Their Work Flows and Methods
  - Get Early Input and Work in the Open
  - Contribute Incremental Bits, Not Large Code Dumps
  - Leave Your Ego at the Door: Do Not Be Thin-Skinned
  - Be Patient, Develop Long-Term Relationships, Be Helpful

Respecting and Encouraging Diversity in OSS

- Diversity in an OSS project can mean many different things:
  - Race and national origin
  - Sex and gender identity
  - National and geographical/regional characteristics
    - Including both language and cultural differences
  - Religious beliefs and political views
  - Acceptance of different opinions and methods
    - How the project should take shape and develop
- *The right thing to do* to accept contributors and reviewers from divergent backgrounds,
- Also leads to a better project due to unleashing more sources of new ideas, approaches, and contributions

Constructing an Open Source Strategy

- Key considerations involved in developing a strategy for integrating open source:



Open Source Program Office

- Serves as a center for functions:
  - Choosing OSS code to be used
  - Facilitating adoption of the code and popularizing its availability and usage
  - Keeping track of usage
  - Auditing compliance considerations
  - Making sure proper information and training is available to all employees, as needed
- Determine the placement of such an office within the organization
  - Should be relatively high in the organizational chart
    - Research and development command structure
    - Legal department
  - Whomever is responsible must be sensitive to both the developer and legal needs

Designing an Open Source Policy

- How much of the product is desired to be OSS?
  - Replacing current closed source components open source ones
- Which license(s) are a good fit to your needs
  - Which licenses can you use in various OSS components you integrate, such as libraries, APIs, etc?
- Enough understanding (and legal help) to set up procedures for using both proprietary and open source code?
- How will you review the code to make sure things are being done properly?
  - Automated license compliance tools?
  - Linux Foundation's Open Compliance Program

Implementing an Open Source Policy

- A policy is worthless without:
  - A structure for ensuring implementation, and
  - Having a dedicated staff tasked with carrying it out
- Training will need to be developed (or outsourced):
  - Developers, Procurement, Legal, Security
- Community participation and outreach can also be critical
  - Who are designated spokespeople?
  - Who monitors developer participation and contributions?
- Clear assignment of responsibility is required in all areas

TODO Group

- Founded in 2014, fostering the adoption and spread of open source software in the corporate business community
- Founding members: Box, Dropbox, Facebook, GitHub, Google, HPE, Khan Academy, Microsoft, Square, Stripe, Twitter, Walmart and Yahoo

TODO Group

- Focuses on:
  - Sharing best practices for running large open source programs.
  - Codifying goals for successfully managed projects.
  - Spreading knowledge about helpful tooling and instrumentation.
  - Listing and describing member projects that are successful examples.
- Every organization has its own unique set of needs, challenges and practices. Sharing allows for faster and better establishment of enabling open source strategies.

TODO Guides

- Guides on building a successful open source program:
  - How to create an open source program
  - Measuring your open source program's success
  - Tools for managing open source programs
- Guides for open source program management:
  - Using open source code
  - Participating in open source communities
  - Recruiting open source developers
  - Starting an open source project
  - Open source reading list
  - Improve your open source development impact
  - Shutting down an open source project
  - Building leadership in an open source community
  - Setting an open source strategy

TODO Case Studies

- Case studies by a diverse group of companies, that describe how they adopted and manage open source
  - Autodesk
  - Capital One
  - Comcast
  - Dropbox
  - Facebook
  - Microsoft
  - Oath
  - Red Hat
  - Salesforce
  - SAP

OpenChain Project

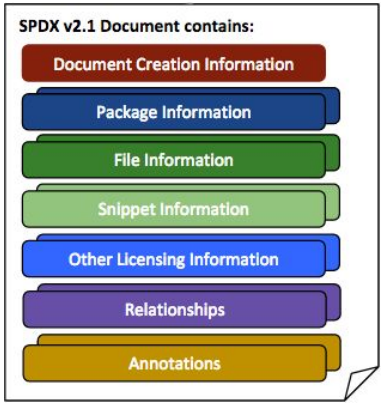
- Established an open set of requirements for effective management of FOSS
- An organization meets this specification of requirements
  - Certified as OpenChain Conforming
  - Receive the right to display the logo indicating that it is fully compliant

FOSSology

- A system/toolkit for measuring and documenting open source license compliance
  - Scan individual files/entire source archives, and identify licenses, copyrights etc.,
  - Document the results, and help identify where information is lacking or is poorly documented
- FOSSology Scanning Features (searches for licenses using)
  - Nomos: regular expression matching + context identification
  - Monk: text-based searching, report back on license variations
  - Copyrights: searching for “copyright” and “(C)” in the text
  - Export Control Codes (ECC): Regular expression searching

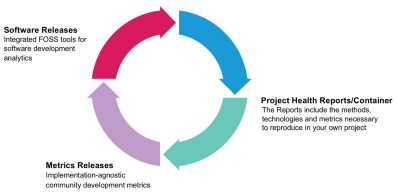
Software Package Data Exchange (SPDX)

- Open standard provides a common format for sharing data about software copyrights, licenses, security information, etc.



Community Health Analytics Open Source Software (CHAOSS)

- Developing tools and metrics for evaluation of the health of open source projects and ecosystems
  - Increasing sustainability over time
  - Making good, informed decisions about how to be involved
- Goals:
  - Establish excellent metrics for measuring community activity, contributions, and health
  - Produce software tooling or doing the analysis
  - Build project health reports



FOSSology

- A system/toolkit for measuring and documenting open source license compliance
  - Scan individual files/entire source archives, and identify licenses, copyrights etc.,
  - Document the results, and help identify where information is lacking or is poorly documented
- FOSSology Scanning Features (searches for licenses using)
  - Nomos: regular expression matching + context identification
  - Monk: text-based searching, report back on license variations
  - Copyrights: searching for “copyright” and “(C)” in the text
  - Export Control Codes (ECC): Regular expression searching

Leadership vs. Control

- OSS projects can be organized in many ways,
  - From very tight control, to an almost anarchistic ecosystem
  - Wide choice of operating methods
- Advantages of leading a project, as opposed to just controlling it
  - Loosen the Chains
    - Good ideas can originate from many different contributors
    - Make the final decision, but listen and give proper consideration to debates in the community
  - Mentoring
    - One level of leadership, or more, including subsystem maintainers
    - Proper mentoring: Empowering people, more efficient workflow, long term vitality
  - Building Trust
    - Without trust, an open source project cannot function
    - Contributors must trust that their submissions will be treated with respect and given reasonably prompt consideration
    - Project leaders must trust that subsystem maintainers are qualified and capable of doing their job

Why Do Many OSS Projects Fail?

- Vast majority of OSS projects do not succeed
- Reasons are varied, but include:
  - Insufficient interest
  - Competition from more established projects that duplicate the intended functionality
  - Poor leadership.
  - Not enough developers.
  - Insufficient funding
  - Insufficient or uninformed attention to licensing issues

Open Source Software Development

Version control - Git and GitHub

Thái Minh Tuấn - Email: minhtuan@ctu.edu.vn

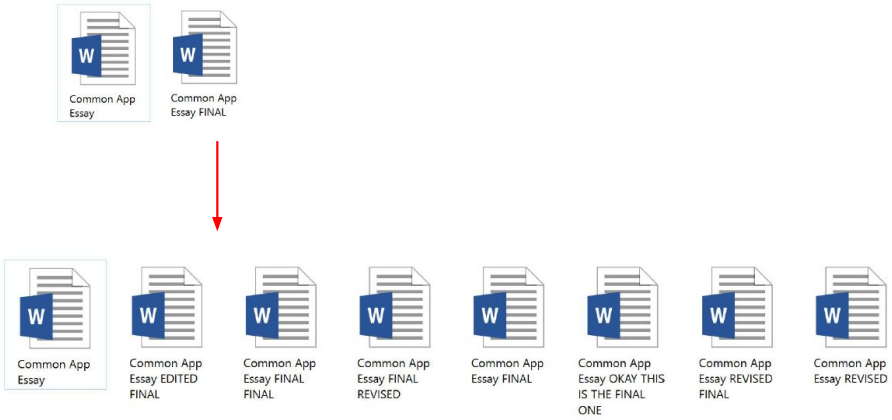
Learning Objectives

- What is version control?
- Why do we need a VCS?
- Centralized version control vs. Distributed version control
- Git: Concept, Terminology, Workflow, etc.
- GitHub

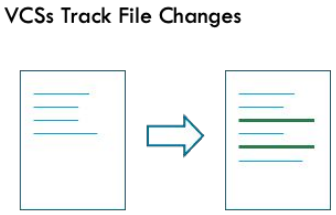
What is version control?

- Version control records changes to a set of files over time
  - This makes it easy to review or obtain a specific version (later)
- Simple example:
  - René writes a paper, using version control: **v1.0**
  - René corrects grammatical mistakes and typos: **v1.1**
  - René discovers new findings and rewrites the paper: **v1.2**
  - René realizes the new findings are wrong: restore **v1.1**
- Who uses version control?
  - Developers (obviously)
  - Researchers
  - Applications (e.g., (cloud-based) word processors)

Why use version control?



Why Do We Need a VCS?



Code is organized within a repository.

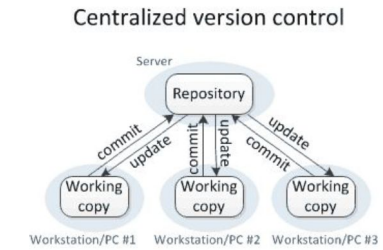
- VCSs tell Us:
  - Who made the change?
    - So you know whom to blame
  - What has changed (added, removed, moved)?
    - Changes within a file
    - Addition, removal, or moving of files/directories
  - Where is the change applied?
    - Not just which file, but which version or branch
  - When was the change made?
    - Timestamp
  - Why was the change made?
    - Commit messages
- Basically, the Five W's

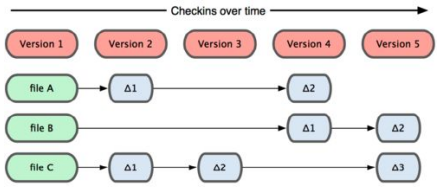
Brief History Of Version Control Software

- First Generation – Local Only
  - SCCS – 1972: Only option for a LONG time
  - RCS – 1982: For comparison with SCCS, see this 1992 forum link
- Second Generation – Centralized
  - CVS – 1986: Basically a front end for RCS
  - SVN – 2000: Tried to be a successor to CVS
  - Perforce – 1995: Proprietary, but very popular for a long time
  - Team Foundation Server – 2005:
    - Microsoft product, proprietary
    - Good Visual Studio integration
- Third Generation – Decentralized
  - BitKeeper – 2000
  - GNU Bazaar – 2005
  - Canonical/Ubuntu
  - Mercurial – 2005
  - Git – 2005
  - Team Foundation Server – 2013

Centralized version control

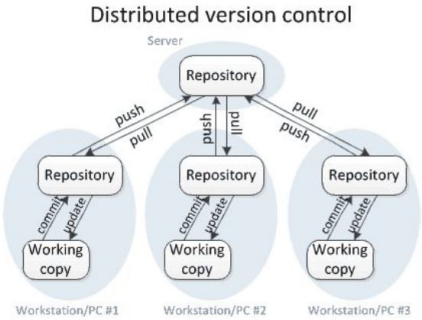
- **One central repository**
- All users commit their changes to the central repository.
  - Each user has a working copy
  - As soon as they commit, the repository gets updated
  - Track version data on each individual file
  - Examples: SVN (Subversion), CVS

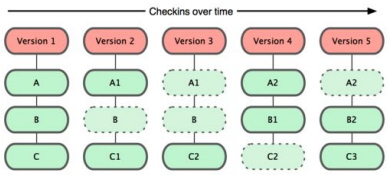




Distributed version control

- **Multiple clones of a repository**
- Each user commits to a local (private) repository
  - All committed changes remain local unless pushed to another repository
  - No external changes are visible unless pulled from another repository
  - Each check-in version of the overall code has a copy of each file in it.
    - Some files change on a given check-in, some do not
    - More redundancy, but faster.
  - Examples: Git, Hg (Mercurial)





About GIT

- Started by Linus Torvalds – 2005
  - Came out of Linux development community
  - Designed to do version control on Linux kernel
- Goals of Git:
  - Speed
  - Support for non-linear development
    - Thousands of parallel branches
    - Branching is cheap and easy!!!!
  - Fully distributed
  - Able to handle large projects efficiently
- Official Site
- Wikipedia
- Initial README commit

(A "git" is a cranky old man. Linus meant himself)

Installing/learning Git

- Git website: http://git-scm.com/
  - Free online book: http://git-scm.com/book
  - Reference page for Git: http://gitref.org/index.html
  - Git tutorial: http://schacon.github.com/git/gittutorial.html
  - Git for Computer Scientists:
    - http://eagain.net/articles/git-for-computer-scientists/
- At command line: (where *verb = config, add, commit*, etc.)

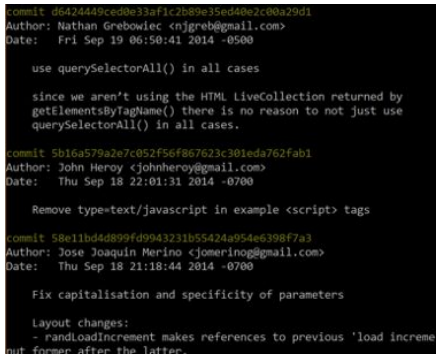
```
git help verb
```


Terminology

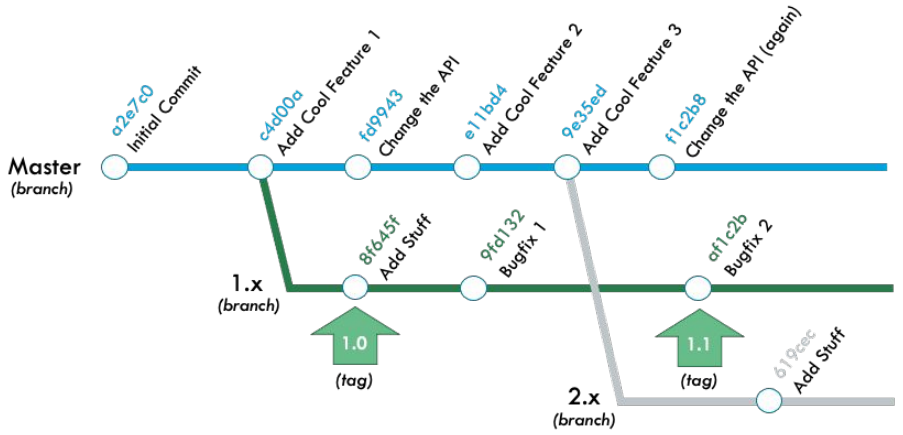
- Branch
  - A history of successive changes to code
  - A new branch may be created at any time, from any existing commit
  - May represent versions of code
    - Version 1.x, 2.x, 3.x, etc.
  - May Represent small bug-fixes/feature development
  - Branches are cheap
    - Fast switching
    - Easy to "merge" into other branches
    - Easy to create, easy to destroy
  - See this guide for best practices
- Commit
  - Set of changes to a repository's files
  - More on this later
- Tag
  - Represents a single commit
  - Often human-friendly
    - Version Numbers
- A Repository may be created by:
  - Cloning an existing one (git clone)
  - Creating a new one locally (git init)

What is a Commit?

- Specific snapshot within the development tree
- Collection of changes applied to a project's files
  - Text changes, file and directory addition/removal, chmod
  - Metadata about the change
- Identified by a SHA-1 Hash
  - Can be shortened to approx. 6 characters for CLI use
    - (e.g., "git show 5b16a5")
  - HEAD – most recent commit
  - ORIG_HEAD – after a merge, the previous HEAD
  - <commit>~n – the nth commit before <commit>
  - e.g., 5b16a5~2 or HEAD~5
  - master@{01-Jan-2018} – last commit on master branch before January 1, 2018



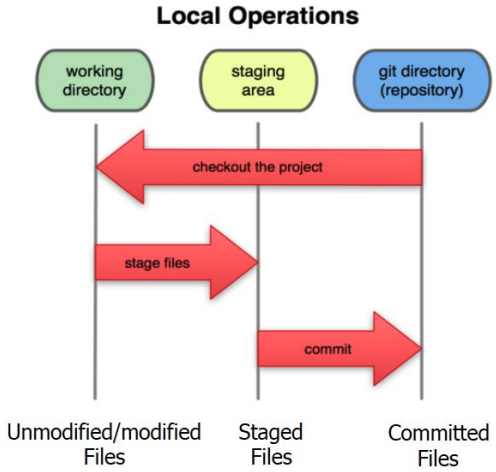
Branches, Commits, and Tags



Terminology

- Working Files
  - Files that are currently on your File System
- The Stage (also called the "index")
  - Staging is the first step to creating a commit
  - The stage is what you use to tell Git which changes to include in the commit
  - File changes must be "added" to the stage explicitly
  - Only changes that have been staged will be committed
- Checkout
  - Replace the current working files with those from a specific branch or commit

Git Workflow - Local operations



Initial Git configuration

- Set the name and email for Git to use when you commit:
  - `git config --global user.name "Bugs Bunny"`
  - `git config --global user.email bugs@gmail.com`
  - You can call `git config -list` to verify these are set.
- Set the editor that is used for writing commit messages:
  - `git config --global core.editor nano`
  - (it is vim by default)

Creating a Git repo

- Two common scenarios: (only do one of these)
- To **create a new local Git repo** in your current directory:
  - `git init`
    - This will create a .git directory in your current directory.
    - Then you can commit files in that directory into the repo
  - `git add filename`
  - `git commit -m "commit message"`
- To **clone a remote repo** to your current directory:
  - `git clone url localDirectoryName`
  - This will create the given local directory, containing a working copy of the files from the repo, and a .git directory (used to hold the staging area and your actual local repo)
- Fork (clone + metadata)
  - Create a completely independent copy of Git repository

Git commands

`git clone url [dir]`	copy a Git repository so you can add to it
`git add file`	adds file contents to the staging area
`git commit`	records a snapshot of the staging area
`git status`	view the status of your files in the working directory and staging area
`git diff`	shows diff of what is staged and what is modified but unstaged
`git help [command]`	get help info about a particular command
`git pull`	fetch from a remote repo and try to merge into the current branch
`git push`	push your new branches and data to a remote repository
Others: `init, reset, branch, checkout, merge, log, tag`	

Add and commit a file

- The first time we ask a file to be tracked, and *every time before we commit a file*, we must add it to the staging area:
  - `git add Hello.java Goodbye.java`
  - Takes a snapshot of these files, adds them to the staging area.
  - In older VCS, "add" means "start tracking this file." In Git, "add" means "add to staging area" so it will be part of the next commit.
- To move staged changes into the repo, we commit:
  - `git commit -m "Fixing bug #22"`
- To undo changes on a file before you have committed it:
  - `git reset HEAD -- filename` #(unstages the file)
  - `git checkout -- filename` #(undoes your changes)
- All these commands are acting on your local version of repo

Viewing/undoing changes

- To view status of files in working directory and staging area:
  - `git status` or `git status -s` (short version)
- To see what is modified but unstaged:
  - `git diff`
- To see a list of staged changes:
  - `git diff --cached`
- To see a log of all changes in your local repo:
  - `git log` or `git log --oneline` (shorter version)
  - `git log -5` (to show only the 5 most recent updates), etc.

An example workflow

```
[rea@attu1 superstar]$ emacs rea.txt
[rea@attu1 superstar]$ git status
  no changes added to commit
  (use "git add" and/or "git commit -a")
[rea@attu1 superstar]$ git status -s
  M rea.txt
[rea@attu1 superstar]$ git diff
  diff --git a/rea.txt b/rea.txt
[rea@attu1 superstar]$ git add rea.txt
[rea@attu1 superstar]$ git status
  #       modified:   rea.txt
[rea@attu1 superstar]$ git diff --cached
  diff --git a/rea.txt b/rea.txt
[rea@attu1 superstar]$ git commit -m "Created new text file"
```

Branching and merging

- Git uses branching heavily to switch between multiple tasks
- To create a new local branch:
  - `git branch name`
- To list all local branches: (* = current branch)
  - `git branch`
- To switch to a given local branch:
  - `git checkout branchname`
- To merge changes from a branch into the local master:
  - `git checkout master`
  - `git merge branchname`

Merge conflicts

- 2 branches modify the same place in the same file
- The conflicting file will contain <<< and >>> sections to indicate where Git was unable to resolve a conflict:

```
<<<<<<< HEAD:index.html
<div id="footer">todo: message here</div>
=======
<div id="footer">
  thanks for visiting our site
</div>
>>>>>>> SpecialBranch:index.html
```

branch 1's version

branch 2's version

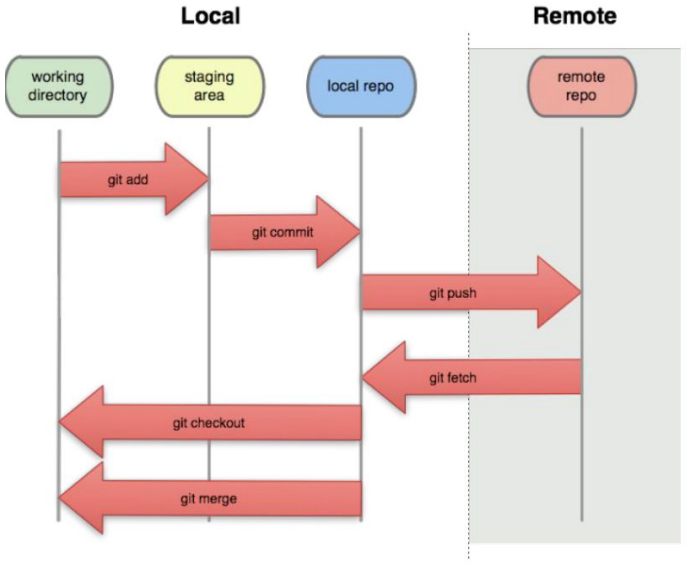
- Find all such sections, and edit them to the proper state (whichever of the two versions is newer / better / more correct)
  - Then, `add` and `commit`

Interaction w/ remote repo

- **Push** your local changes to the remote repo
- **Pull** from remote repo to get most recent changes
  - (fix conflicts if necessary, add/commit them to your local repo)

- To fetch the most recent updates from the remote repo into your local repo, and put them into your working directory:
  - `git pull origin master`
- To put your changes from your local repo in the remote repo:
  - `git push origin master`

Git Workflow



Common Industry Programmer Workflow

1. Choose a bug/feature to work on
2. Checkout the appropriate branch
3. Pull the latest additions from the main repository
4. Create a branch for the bug/feature request
5. Do the coding necessary & commit
6. Push the changes to your remote
7. Create a pull request to main repo

The pull request is where other programmers review your code & make comments/suggestions

If changes are needed to your code, then repeat the process.

Never commit directly to the main repo

Practical Guidelines - DO's

- Make small, incremental commits (within reason) are good
  Avoid monolithic commits at all cost
- Use a separate branch for each new bug/feature request
- Write nice commit messages
  - Otherwise, the commit log is useless
- Use a .gitignore to keep cruft out of your repo
- Search engines are your friend
  - Someone else has had the same question/mistake/situation as you, and they have already asked the question online. It's probably on StackOverflow

Practical Guidelines - Dont's

- Do not commit commented-out debug code.
  - It's messy. It's ugly. It's unprofessional.
- Do not mix your commits.
  - E.g., Don't commit two bug-fixes at the same time
- Do not commit sensitive information
  - Passwords, database dumps, etc.)
- Do not commit whitespace differences, unless it is specifically needed
- Do not commit large binaries

Practical Guidelines - Dont's

- Pro Git – free Apress ebook
- Visualizing Git histories
- Wikipedia on Version Control, with definitions
- Git Cheat Sheet
- Git Commit Etiquette for Junior Devs
- https://www.codeschool.com/learn/git
- Free course to walk you through basic Git concepts

GitHub (1/2)

- Git began as an offshoot of the Linux kernel development community
  - Initially created by Linus Torvalds himself
- However, it was quickly realized it could be used for any project that had similar needs:
  - A large group of contributors
  - A widely dispersed community of contributors
  - A very open development method with frequent releases
- Use of Git grew explosively after the founding of GitHub in 2008
  - 3 years after the creation of Git.
- GitHub.com is a site for online storage of Git repositories
  - You can create a remote repo there and push code to it
  - Many open source projects use it

GitHub (2/2)

- Before GitHub projects needed to have their own servers to host repositories
  - Need good amount of knowledge to setup, administer and secure and protect the integrity of repositories.
  - It is now pretty easy to get a project rolling quickly.
- In 2018, GitHub was acquired by Microsoft. In early 2020, GitHub had over 40 million users and 100 million repositories!
- Question: Do I always have to use GitHub to use Git?
  - No! You can use Git locally for your own purposes.
  - Or you or someone else could set up a server to share files
  - There are other sites that offer similar services, including:
    - GitLab, GitKraken, Launchpad

Public vs Private Repositories

- Private Repositories
  - Only selected collaborators can see the repository, or clone it (make a local copy), or download its contents in a variety of forms
  - The owner must specifically authorize each collaborator
- Public Repositories
  - Anyone given the proper link, can copy, clone or fork the repository, or download its contents
  - However, unless the owner authorizes them as a collaborator, one does not have permission to upload or make modifications

Open Source Software Development

Linux Kernel Development

Thái Minh Tuấn - Email: minhtuan@ctu.edu.vn

Learning Objectives

- Linux Kernel Development Process
- Release Cycle - Kernel Trees - Patches
- Working on Linux community
- Building and Installing Your First Kernel
- Writing Your First Kernel Patch
- Testing - Debugging
- Continue Your Kernel Journey

Linux Kernel Development Process

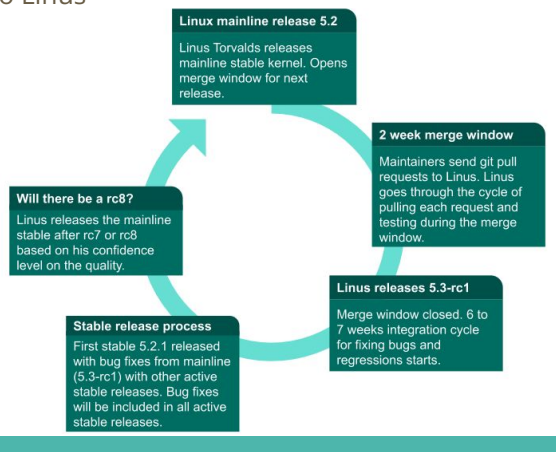
- The result of collaborative development efforts from developers across the globe
  - Over 13,000 kernel developers (2020)
- *Patches*: Small incremental changes
  - Add new features, make enhancements, and fix bugs
- *Releases*: New versions of Linux kernel
- Linux kernel development process:
  - 24-hour, 7 days a week, and 365 days of continuous process
  - New release once every 2+ months (10 to 11 weeks)
    - Several stable and extended stable releases once a week
  - Time-based rather than feature-based
    - Not held up for features, no set date for releases
  - New development and current release integration cycles run in parallel

Release Cycle (1/2)

- Linus Torvalds releases a new kernel and opens a *2-week merge window*
  - Pulls code for the next release from subsystem maintainers
  - Subsystem maintainers send signed git pull requests to Linus either during the merge window or before
  - All major new development is added to the kernel
  - 10,000+ change sets (patches) get pulled into Linus's tree
- At the end of merge window, Linus releases the first release candidate (rc1)
  - Release cycle moves into a bug fixes-only mode
  - Once a week, a new rc comes out: rc2, rc3, rc4, etc.
    - Until all major bug fixes and regressions (if any) are resolved

Release Cycle (2/2)

- New cycle begins with a *3-week quiet period*: Starts a week before the release, and continues through the 2-week merge window
  - Maintainers/contributors are busy getting their trees ready to send pull requests to Linus



Kernel Trees

- Mainline kernel tree
  - Maintained by Linus Torvalds
  - Releases mainline kernels and RC releases
- Stable tree
  - Maintained by Greg Kroah-Hartman
  - Consists of stable release branches
- Linux-next tree
  - The integration tree maintained by Stephen Rothwell
  - Code from a large number of subsystem trees gets pulled into this tree periodically and then released for integration testing

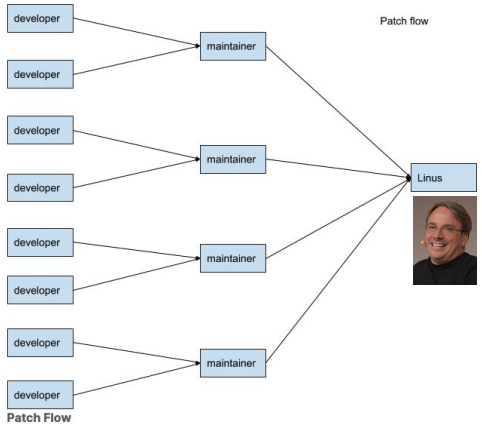
Active Kernel Releases

- Release Candidate (RC)
  - Mainline kernel pre-releases that are used for testing new features
  - Developers test these releases for bugs and regressions
- Stable
  - Bug fix-only releases. Released once a week, or on an as needed basis
  - After mainline kernel released, it moves into stable mode
  - Bug fixes for a stable kernel are backported from the mainline kernel
  - Only maintained for a few mainline release cycles
- Long-term
  - Stable releases selected for long-term maintenance
    - Provide critical bug fixes for older kernel trees
  - Maintained for a longer period
    - Allow multiple vendors collaborate on a specific kernel release
      - Plan on maintaining for an extended period of time

https://www.kernel.org/category/releases.html

Subsystem Maintainers

- Each major subsystem has its own tree and designated maintainer(s)
  - MAINTAINERS file: List of subsystems and their maintainers
- Development process itself happens entirely over emails
  - Almost every kernel subsystem has a mailing list
  - Contributors send patches to mailing lists through emails
  - Contributions are then discussed through emails
  - Maintainers send an email response to contributors when they accept and commit patches



"It is extremely amazing how the Linux community, which is worldwide, does all its work in email."

Patches

- Small incremental changes made to the kernel
  - Developers send to the kernel mailing lists through email
- Independent modification that stands on its own
  - Cannot break the kernel build
  - Easier to isolate regressions
  - Complex changes to the kernel are thus split into smaller chunks
  - Find an existing compile warning while making a code change
    - Fix it independently in a separate patch instead of combining it with your code change
- Maintainers have their personal preferences on how granular the patch splitting should be for their subsystems
  - Giving feedback on their preferences during the patch review

Create and Apply Git Patch Files

- Modify the user-specific Git configuration file `~/.gitconfig`
- Create Git patch files
  - `git format-patch <branch> <options>`
  - Example: Create Git patch for 3 topmost commits
    - `git format-patch --pretty=fuller -3`
- Apply Git patch files
  - `git am <patch_file>`

Patch Tags

- Acked-by: Used by the maintainer of the affected code when that maintainer neither contributed to, nor forwarded the patch
- Reviewed-by: Indicates that the patch has been reviewed by the person named in the tag
- Reported-by: Gives credit to people who find bugs and report them.
- Tested-by: Indicates that the patch has been tested by the person named in the tag
- Suggested-by: Give credit for the patch idea to the person named in the tag
- Fixes: Indicates that the patch fixes an issue in a previous commit referenced by its Commit ID, allows us track where the bug originated

Patch Email Subject Line Conventions

- [PATCH] prefix is used to indicate that the email consists of a patch
  - [PATCH v4] is used to indicate that the patch is the 4th version of this specific change that is being submitted
- [PATCH RFC] or [RFC PATCH] indicates the author is requesting comments on the patch
  - RFC stands for "Request For Comments"
- NOT unusual for a patch to go through a few revisions before it gets accepted
  - An artifact of collaborative development
  - The goal is to get the code right and not rush it in

Patch Version History

- Include the patch version history when sending a re-worked patch
- The patch revision history on what changed between the current version and the previous version is added between the "---" and the "start of the diff" in the patch file
  - Any text that is added here gets thrown away and will not be included in the commit when it is merged into the source tree
  - It is good practice to include information that helps with reviews and doesn't add value to the commit log
  - Check mailing lists to get a feel for what kind of information gets added
- Do not send new versions of a patch as a reply to a previous version
- Patch example with version history for the v2 version

Linux Kernel Contributor Covenant Code of Conduct

- Linux kernel community abides by the Linux Kernel Contributor Covenant Code of Conduct
  - Contributors and maintainers pledge to foster an open and welcoming community to participate in and become part of
- Linux kernel is often a labor of love of a community that truly thinks independently together
  - We value diversity of thought in solving technical problems
- How to conduct yourself as a contributor and participant in the community
  - Contributor Covenant Code of Conduct
  - Linux Kernel Contributor Covenant Code of Conduct Interpretation
- It is important to understand that when you send a patch with your sign-off, you are agreeing to abide by the code of conduct outlined in these two documents

Linux Enforcement Statement

- Linux Kernel is provided under the terms of the GNU General Public License version 2 only (GPL-2.0)
  - With an explicit syscall exception described here
- Linux Kernel Enforcement Statement
  - How our (Linux Kernel) software is used and how the license for our software is enforced
  - Balance the enforcement actions with the health and growth of the Linux kernel ecosystem
- You are encouraged to read the statement to understand the intent and spirit of the statement and to get a better understanding of our community
  - *Working together, we will be stronger*

Configuring Your Development System (1/3)

- Getting Your System Ready
  - A regular laptop is a good choice for a basic development system
  - Install the Linux distribution of your choice
  - Minimal requirements to compile the Kernel
    - `sudo apt update`
    - `sudo apt-get install build-essential vim git cscope libncurses-dev libssl-dev bison flex`
  - Configuring email client for sending patches and responding to emails
    - Highly recommend using git send-email

Configuring Your Development System (2/3)

- Git Email Configuration
  - Configuring `git-email` to send patches is easy using the `sendemail` configuration option, once you have your smtp server configuration

```
[sendemail]
        smtpserver     = smtp.gmail.com
        smtpserverport = 587
        smtpencryption = tls
        smtpuser       = <username>
        smtppass       = <password>
```

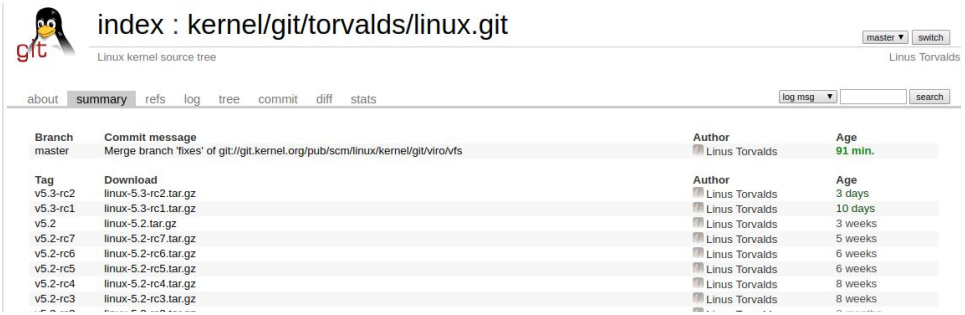
  - Place the above configuration in your `.gitconfig` file
  - Running `git send-email mypatch.patch` is all you have to do to send patches
    - `mypatch.patch` is generated by `git format-patch` command

Configuring Your Development System (3/3)

- Email Client Configuration
  - Refer to *Email clients info for Linux* for more details
  - Things to remember
    - Bottom post
    - Inline post
    - No HTML format
    - No signatures
    - No attachments
  - Refer to the Examples in *this document* for more information

Exploring Linux Kernel Sources

- The latest stable and mainline releases hosted on *The Linux Kernel Archives* web page
- Access the Linux mainline to explore the kernel sources



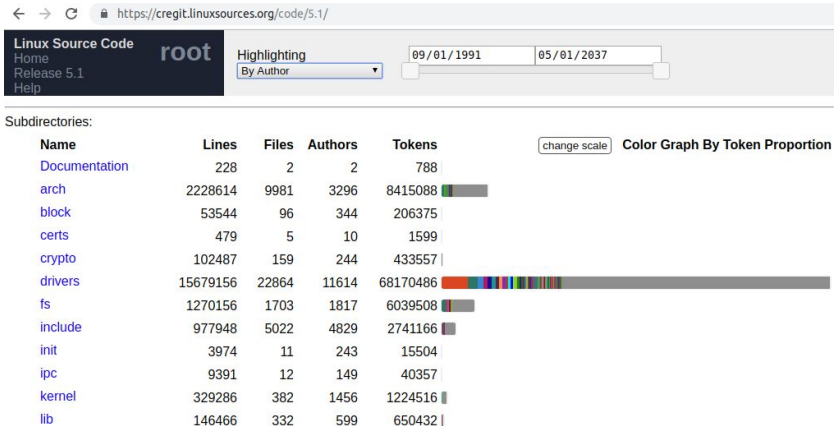
Cloning the Linux Mainline

- Create a new directory named linux_mainline and populate it with the sources
  - `cd /linux_work`
  - `git clone \ git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git linux_mainline`
  - `cd linux_mainline; ls -h`

```
arch      CREDITS       drivers  ipc      lib          mm        scripts   usr
block     crypto        fs       Kbuild   LICENSES     net       security  virt
certs     cscope.out    include  Kconfig  MAINTAINERS  README    sound
COPYING   Documentation init     kernel   Makefile     samples   tools
```

What Is in the Root Directory?

- This screenshot presents a view of the Linux 5.1 release on cregit-Linux



Name	Lines	Files	Authors	Tokens
Documentation	228	2	2	788
arch	2228614	9981	3296	8415088
block	53544	96	344	206375
certs	479	5	10	1599
crypto	102487	159	244	433557
drivers	15679156	22864	11614	68170486
fs	1270156	1703	1817	6039508
include	977948	5022	4829	2741166
init	3974	11	243	15504
ipc	9391	12	149	40357
kernel	329286	382	1456	1224516
lib	146466	332	599	650432

Exploring the Sources

- You are encouraged to explore the sources. Take a look at below files, you will be using them in your everyday kernel development life.
  - Makefile and MAINTAINERS files in the main directory
  - scripts/get_maintainer.pl and scripts/checkpatch.pl
- Play with `cregit` or `git log` to look at the history of source files in each of the kernel areas
- Look at individual commits and generate a patch or two using
  - `git format-patch -1 <commit ID>`

Building and Installing Your First Kernel

- Cloning the Stable Kernel Git
  - `git clone \ git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git linux_stable`
  - `cd linux_stable`
  - `git branch -a | grep linux-5`
    - `remotes/origin/linux-5.12.y`
    - `remotes/origin/linux-5.11.y`
    - `remotes/origin/linux-5.10.y`
  - `git checkout linux-5.12.y`

Building and Installing Your First Kernel

- Generate a kernel configuration file based on the current configuration
  - Using the distribution configuration file is the safest approach for the very first kernel install on any system
    - Copying from `/proc/config.gz` or `/boot`
  - `ls /boot`
  - `cp /boot/<config-5.0.0-21-generic> .config`

```
config-5.0.0-20-generic        memtest86+.bin
config-5.0.0-21-generic        memtest86+.elf
efi                            memtest86+_multiboot.bin
grub                           System.map-5.0.0-20-generic
initrd.img-5.0.0-20-generic    System.map-5.0.0-21-generic
initrd.img-5.0.0-21-generic    vmlinuz-5.0.0-20-generic
lost+found                     vmlinuz-5.0.0-21-generic
```

Building and Installing Your First Kernel

- Generate a kernel configuration file based on the current configuration
  - `make olddefconfig`
- Another way: Creates a configuration file based on the list of modules currently loaded on your system
  - `lsmod > /tmp/my-lsmod`
  - `make LSMOD=/tmp/my-lsmod localmodconfig`
- It is time to compile the kernel
  - `make -j3 all`
  - `j` option specifies the number of jobs (make commands) to run simultaneously
- Compiling a single source file: `make path/file.o`
- Compiling at the directory level: `make path`

Linux Kernel Configuration

- Linux kernel is completely configurable
- Drivers can be:
  - Disabled
  - Built into the kernel (vmlinux image) to be loaded at boot time
  - Built as a module to be loaded as needed using `modprobe`
- Should configure drivers as modules, to avoid large kernel images
  - Modules (.ko files) can be loaded when the kernel detects hardware that matches the driver
- `modprobe`
  - Add/remove a module from the Linux kernel
- `sysctl`
  - List/modify kernel parameters at runtime (listed under `/proc/sys/`)

Writing Your First Kernel Patch

- Creating a new Branch in the linux_mainline repository
  - `git checkout -b first-patch`
- Update the kernel
  - `git fetch origin`
- Making changes to a driver
  - Run `lsmod` to see the modules loaded on your system, and pick a driver to change
  - One driver that's included in all VM images is the `e1000` driver, the Intel ethernet driver
  - Run `git grep` to look for `e1000` files
    - `git grep e1000 -- '*Makefile'`

Writing Your First Kernel Patch

- Make a small change to the probe function of the e1000 driver
  - `nano drivers/net/ethernet/intel/e1000/e1000_main.c`

```
static int e1000_probe(struct pci_dev *pdev, const struct
pci_device_id *ent) {
        struct net_device *netdev;
        struct e1000_adapter *adapter;
        struct e1000_hw *hw;
        printk(KERN_DEBUG "I can modify the Linux kernel!\n");
        static int cards_found = 0;
```

- Compile, install, test your changes
  - `make -j3`
  - `sudo make modules_install install`
  - `dmesg | less`

Sending a Patch for Review

- Committing changes, and view your commit
  - `git commit -s -v`
  - `git show HEAD`
  - `git log --pretty=oneline --abbrev-commit`
- The `get_maintainer.pl` script tells you whom to send the patch to
  - `git show HEAD | scripts/get_maintainer.pl`
- Creating a patch
  - `git format-patch -1 <commit ID> --to=<maintainer email> --cc=<maintainer email>`
- Sending patches
  - `git send-email <patch_file>`

Applying Patches

- Linux kernel patch files are text files that contain the differences from the original source to the new source
  - `git apply --index file.patch`

The Review Process

- Your patch will get comments from reviewers with suggestions for improvements
  - In some cases, learning to know more about the change itself
- Please be patient and wait for a minimum of one week before requesting a response.
  - During merge windows and other busy times, it might take longer than a week to get a response
  - Make sure you sent the patch to the right recipients
- Best Practices for Sending Patches

Testing Open Source Software

- Testing is very important when it comes to any software, not just the Linux kernel
- Ensuring software is stable without regressions before the release helps avoid debugging and fixing customer/user-found bugs after the release
  - It costs more in time and effort to debug and fix a customer-found problem
- In the open source development model, developers and users share the testing responsibility
- Users and developers that are not familiar with the new code may be more effective at testing a new piece of code than the original author of that code

Automated Build Bots and Continuous Integration Test Rings

- Maintainers and developers can request their repositories to be added to the linux-next integration tree and the 0-day build bot
- 0-day build bot can pull in patches and run build tests on several configurations and architectures
  - Helpful in finding compile errors and warnings that might show up on other architectures that developers might not have access to
- Continuous Integration (CI) rings are test farms that host several platforms and run boot tests and Kernel Self-tests on stable, linux-next, and mainline trees
  - Kernel CI Dashboard
  - 0-Day - Boot and Performance issues
  - 0-Day - Build issues
  - Linaro QA
  - Buildbot

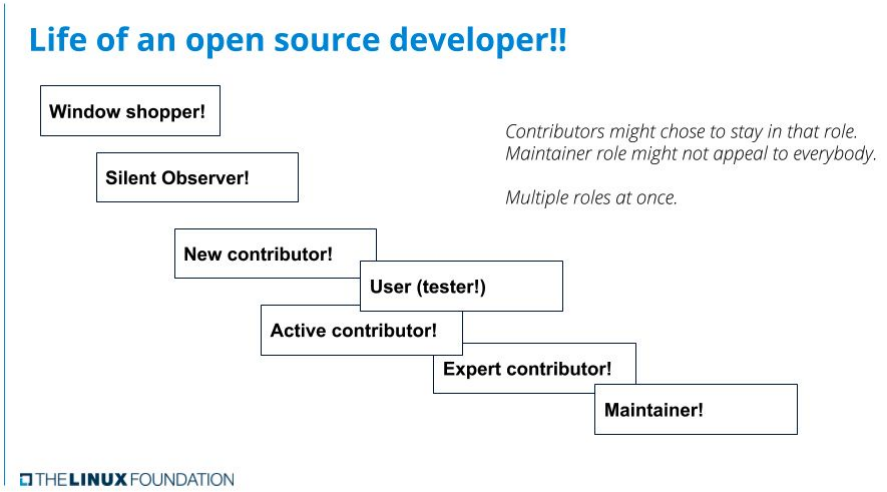
Kernel Debugging (1/2)

- Debugging is an art, and not a science. There is no step-by-step procedure or a single recipe for success when debugging a problem
- Asking the following questions can help to understand and identify the nature of the problem and how best to solve it:
  - Is the problem easily reproducible?
  - Is there a reproducer or test that can trigger the bug consistently?
  - Are there any panic, or error, or debug messages in the dmesg when the bug is triggered?
  - Is reproducing the problem time-sensitive?

Kernel Debugging (2/2)

- What's in a Panic Message?
  - Debugging Analysis of Kernel panics and Kernel oopses using System Map
  - Understanding a Kernel Oops!
- Decode and Analyze the Panic Message
  - decode_stacktrace: make stack dump output useful again
- Use Event Tracing to Debug
  - Event Tracing page in the Linux Kernel Documentation
- Additional resources
  - Hunt bugs
  - Bisecting a bug
  - Dynamic debugging
  - Contributors to the Linux Kernel.
  - Who Made That Change and When: Using Cregit for Debugging

Life of an Open Source Developer



How does one become a maintainer?

- There are many paths to choose from. A few paths that lead you there
  - Write a new driver
  - Adopt an orphaned driver or a module (Hint: Check the MAINTAINERS file).
  - Clean up a staging area driver to add it to the mainline
  - Work towards becoming an expert in an area of your interest by fixing bugs, doing reviews, and making contributions.
  - Find a new feature or an enhancement to an existing feature that is important to the kernel and its ecosystem
- Remember that new people bringing new and diverse ideas make the kernel better. We, as a community, welcome and embrace new ideas!

Continue Your Kernel Journey

- Participate in the Stable Release Process
  - Subscribe to the Stable release mailing list
  - Stable release maintainers send stable patches for testing with information on where to download the patch from
  - Download the patches or clone repository specified in the email
  - Compile and apply the patches
- Enhance and Improve Kernel Documentation

Continue Your Kernel Journey

- Contribute to the Kernel - Getting Started
  - Subscribe to the Linux Kernel mailing list for the area of your interest
  - Follow the development activity reading the Linux Kernel Mailing List Archives
  - Join the #kernelnewbies IRC channel on the OFTC IRC network
  - Join the #linux-kselftest, #linuxtv, #kernelci, or #v4l IRC channels on freenode
  - Find spelling errors in kernel messages
  - Static code analysis error fixing
  - Bug reports and fixing
  - Etc.
- Linux Foundation Events Calendar
- LF Live: Mentorship Series

Open Source Software Development

Continuous Integration (CI) - Continuous Delivery (CD)

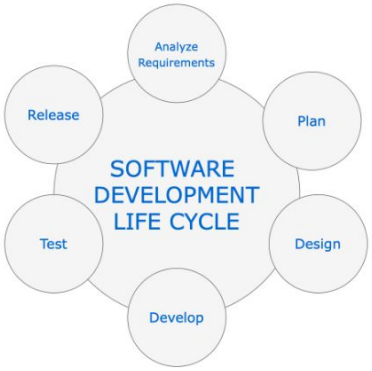
Thái Minh Tuấn - Email: minhtuan@ctu.edu.vn

Learning Objectives

- Explain the fundamental concepts of DevOps, Continuous Integration (CI), Continuous Delivery (CD), and Continuous Deployment
- Understand the basics of Continuous Delivery pipelines
- Distinguish different tools available for CI/CD and explain why Jenkins is a good choice

Software Development Life Cycle

- Software development follows a flow:
  - Identifying new features
  - Planning
  - Doing the actual development, committing the source code changes
  - Running builds and tests (unit, integration, functional, acceptance, etc.)
  - Deploying to production



Continuous Integration (CI)

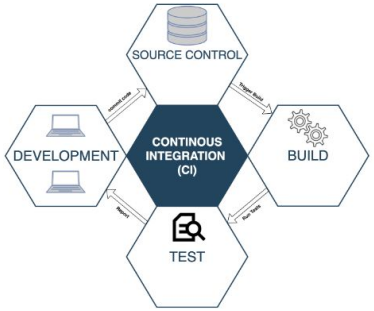
- Continuous Integration is an agile engineering practice originating from the extreme programming methodology. It primarily focuses on automated build and test for every change committed to the version control system by the developers.

"Continuous Integration (CI) is a software development practice where members of a team integrate their work frequently; usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible".

Martin Fowler

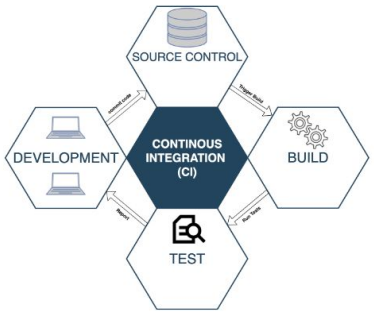
CI Implementation - Requirements (1/2)

- A version control system
  - It stores all the source code checked in by the teams, and acts as the single source of truth.
- Automated build and unit test
  - It is not sufficient if the code written by a developer works only on his/her machine.
  - Every commit that makes it to the version control system should be built and tested by an independent continuous integration server.



CI Implementation - Requirements (2/2)

- Feedback
  - Developers should get feedback on their commits.
  - Anytime a developer's change breaks the build, they can take the necessary action (example: email notifications).
- Agreement on ways of working
  - It is important that everyone on the team follows the practice of checking-in incremental changes rather than waiting till they are fully developed.
  - Their priority should be to fix any build issues that may arise with the checked-in code.



Continuous Delivery (1/2)

- A logical extension of Continuous Integration.
- Automates the full application delivery process by taking any change in code (new features, bug fixes, etc.) all the way from development (code commit) to deployment (to environments such as staging and production).
- Ensures that you are able to release new changes to your customers quickly in a reliable and repeatable manner.

Continuous Delivery is a software development discipline where you build software in such a way that the software can be released to production at any time.

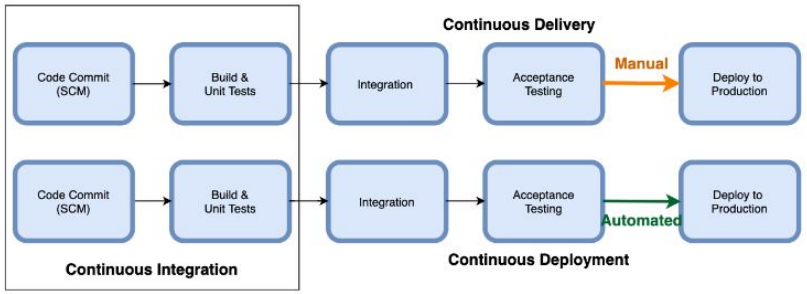
Continuous Delivery (2/2)

- To achieve continuous delivery you need:
  - A close, collaborative working relationship between everyone involved in delivery (often referred to as a DevOps culture)
  - Extensive automation of all possible parts of the delivery process, usually using a deployment pipeline
- Continuous Delivery practices will make:
  - Your overall release process painless
  - Reduce the time to market for new features, and increase the overall quality of software thereby leading to greater customer satisfaction
  - It can also significantly reduce your software development costs as your teams will prioritize releasing new features over debugging defects

Continuous Deployment

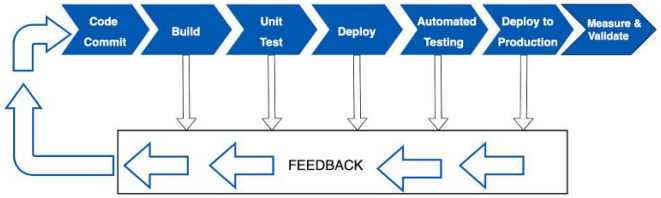
- While Continuous Delivery gives you the capability to deploy to production frequently, it does not necessarily mean that you are automating the deployment. You may need manual approval prior to deploying to production, or your business may not want to deploy frequently.
- Continuous Deployment, however, is an automated way of deploying your releases to production. You need to be doing continuous delivery in order to be able to perform automated deployment. Companies like Netflix, Amazon, Google, and Facebook automatically deploy to production multiple times a day.

Continuous Deployment



Deployment Pipelines

- Automate all the stages of your software delivery process:
  - A developer committing source code change into a version control repository.
  - The CI server detects the new commit, compiles the code, and runs unit tests.
  - The next stage is deploying the artifacts (files generated from a build) to staging or a feature test environment where you run additional functional, regression, and acceptance tests.
  - Once all the tests are successful, you are ready to deploy into production.
  - In case of failure during any stage, the workflow stops and an immediate feedback is sent back to the developer.



Benefits of CI/CD

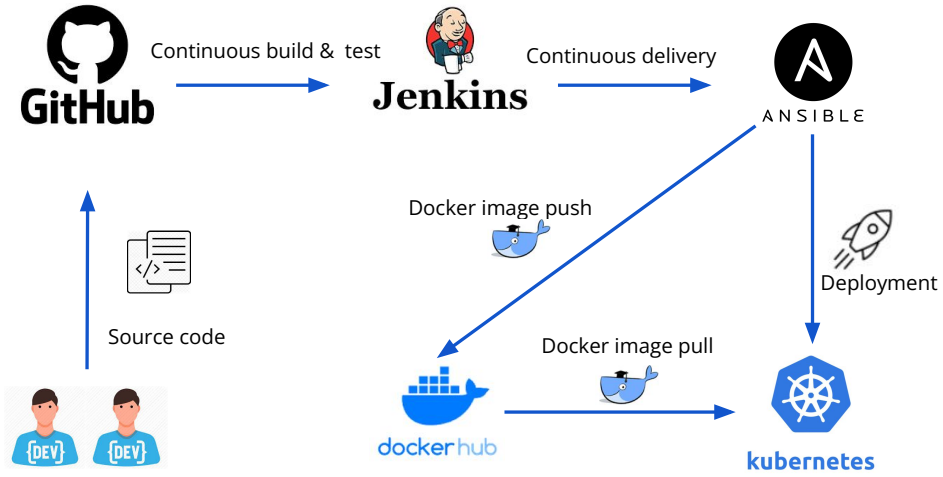
- Faster time to market
- Reduced risk
- Shorter review time
- Better code quality
- Smoother path to production
- Faster bug fixes
- Efficient infrastructure
- Measurable progress
- Tighter feedback loops
- Collaboration and communication
- Maximized creativity

https://www.jetbrains.com/teamcity/ci-cd-guide/benefits-of-ci-cd/

Tools for Deployment Pipeline

- To automate the various stages of your deployment pipeline, you will need multiple tools:
  - A version control system such as Git to store your source code
  - A Continuous Integration (CI) tool such as Jenkins to run automated builds
  - Test frameworks such as xUnit, Selenium, etc., to run various test suites
  - A binary repository such as DockerHub or Artifactory to store build artifacts
  - Configuration management tools such as Ansible
  - A single dashboard to make the progress visible to everyone
  - Frequent feedback in the form of emails, or Slack notifications.

CI/CD pipeline example

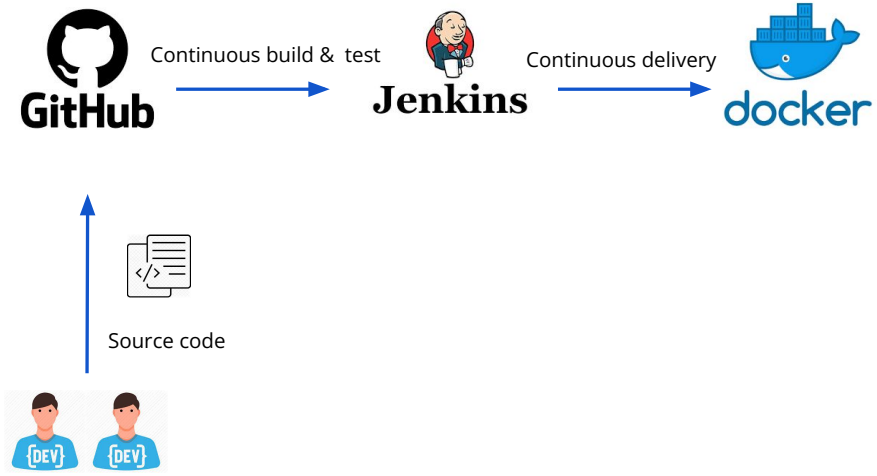


Applying CI/CD pipeline for a Python Flask project

- Flask : A micro web framework written in Python

- Automatically Dockerize a Flask Project on GitHub Push with Jenkins
  - Docker: A set of platform use OS-level virtualization to deliver software in packages called containers
  - GitHub: Internet hosting for version control using Git
  - Jenkins: A free and open source automation server

Applying CI/CD pipeline for a Python Flask project



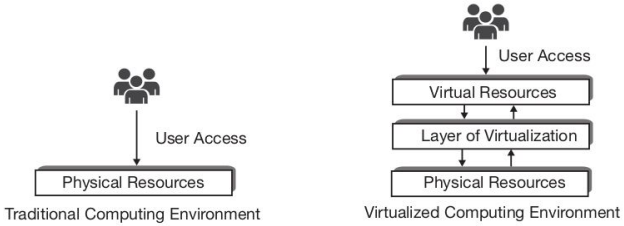
Open Source Software Development

Resource Virtualization

Thái Minh Tuấn - Email: minhtuan@ctu.edu.vn

What Is Virtualization

- The most significant among several enabling technologies of cloud computing
- The representation of physical computing resources in simulated form having made through the software
  - The logical separation of physical resources from direct access of users to fulfill their service needs. Decouples the physical computing resources from direct access of users.

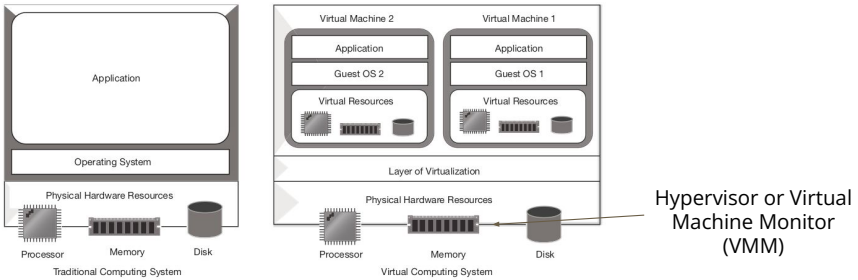


What Is Virtualization

- Any kind of computing resources can be virtualized
  - Processor, memory, storage, network devices, peripheral devices (like keyboard, mouse, printer), etc.
  - In case of core computing resources, a virtualized component can only be operational when a physical resource empowers it from the back end
- The simulated devices produced through virtualization may or may not resemble the actual physical components (in quality, architecture or in quantity)
- Virtual computers (virtual machines) can be built using virtual computing resources produced by virtualization
- Business benefits: Lower hardware cost and improvement in server utilization ratio

Machine Or Server Level Virtualization

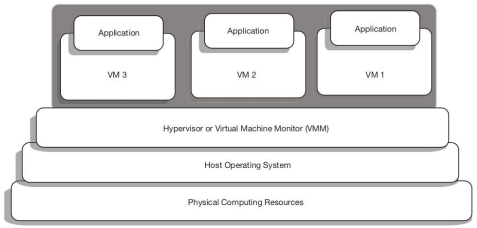
- Machine/server virtualization is the concept of creating virtual machine (guest systems) on actual physical machine (host system)
  - Virtual machines running over a single host system, remain independent of each other
  - Operating systems are installed into those virtual machines

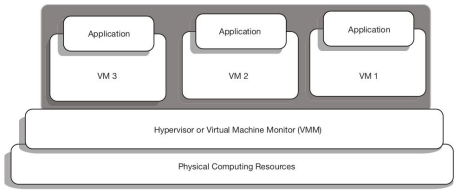


Machine Virtualization Techniques (1/2)

- Hosted approach (Type-2 hypervisor / Hosted hypervisor)
  - VMWare Workstation, VirtualBox, Microsoft Virtual PC, .etc
  - An operating system is first installed on the physical (host) machine
  - The hypervisor is then installed over host OS
  - Compatible for a wide variety of hardware platform, degrade the performance of the virtual machines
- Bare-metal approach: Removal of the Host OS (Type-1 hypervisor / Native Hypervisor)
  - VMware's ESX and ESXi Servers, Microsoft's Hyper-V, Xen
  - The hypervisor is directly installed over the physical machine
  - Provide better performance, advanced features for resource and security management
  - Have limited hardware support and cannot run on a wide variety of hardware platform

Machine Virtualization Techniques (2/2)





Hypervisor Or Virtual Machine Monitor

- Presents a virtual operating platform before the guest systems
- Monitors and manages the execution of guest systems and the VMs
- Allows the sharing of the underlying physical resources among VMs
- Hypervisor-Based Virtualization Approaches:
  - Full Virtualization (native virtualization)
  - Para-Virtualization or OS-Assisted Virtualization
  - Hardware-Assisted Virtualization

Full Virtualization

- The hypervisor fully simulates or emulates the underlying hardware. Virtual machines run over these virtual set of hardware.
- The guest operating systems assume that they are running on actual physical resources and thus remain unaware that they have been virtualized.
- This enables the unmodified versions of available operating systems (like Windows, Linux and else) to run as guest OS over hypervisor.
- In full virtualization technique, the guest operating systems can directly run over hypervisor.

Para(beside/alongside)-Virtualization (1/2)

- Para-virtualization requires hypervisor-specific modifications of guest operating systems.
- A portion of the virtualization management task is transferred (from the hypervisor) towards the guest operating systems. Normal versions of available operating systems need special modification (porting) for this capability inclusion.
  - Each guest OS needs to have prior knowledge that it will run over the virtualized platform, has to know on which particular hypervisor they will have to run.
- Best known example of para-virtualization hypervisor is the open-source Xen project which uses a customized Linux kernel

Para(beside/alongside)-Virtualization (2/2)

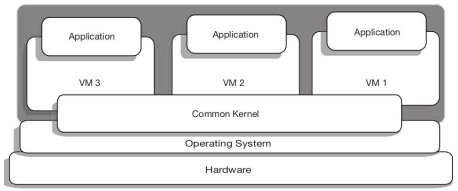
- Reduces the virtualization overhead of the hypervisor as compared to the full virtualization
- The system is not restricted by the device drivers provided by the virtualization software layer
  - Unmodified versions of available operating systems (like Windows or Linux) are not compatible with para-virtualization hypervisors.
  - Modifications are possible in Open-source operating systems (like Linux) by the user. But for proprietary operating systems (like Windows), it depends upon the owner
- Security is compromised in this approach as the guest OS has a comparatively more control of the underlying hardware

Hardware-Assisted Virtualization

- Inspired by software-enabled virtualization, hardware vendors later started manufacturing devices tailored to support virtualization
  - Intel and AMD started this by including new virtualization features in their processors
- The AMD-Virtualization (AMD-V) and Intel Virtualization Technology (Intel-VT): allows some privileged CPU calls from the guest OS to be directly handled by the CPU
- Hardware-assisted virtualization requires explicit features in the host machine's CPU.

Operating System Level Virtualization

- No hypervisor is used and the virtual servers are enabled by the kernel of the operating system of physical machine
- The kernel of the operating system installed over physical system is shared among all of the virtual servers running over it
- Create multiple logically-distinct user-space instances (virtual servers) over a single instance of an OS kernel
- Examples: FreeBSD's jail, Linux VServer (Docker/LXC), OpenVZ



- Advantages: lighter in weight since all of the virtual servers share a single instance of an OS kernel.
- Limitations: All VMs have to use the OS.

Major Server Virtualization Products And Vendors

- VMware vSphere
- Citrix XenServer
- Microsoft Hyper-V Server
- Oracle VM VirtualBox
- KVM
- etc.

High-level Language Virtual Machine

- Also known as application VM or process VM
- Against the idea of conventional computing environment where a compiled application is firmly tied to a particular OS and ISA (instruction set architecture)
- Use the porting of compiler by rendering HLLs to intermediate representation targeted towards a*bstract machines*. The abstract machine then translates the intermediate code to physical machine's instruction set
  - Java Virtual Machine (JVM)
  - Microsoft's Common Language Runtime (CLR)

Advantages Of Virtualization

- Better utilization of existing resources
- Reduction in hardware cost
- Reduction in computing infrastructure costs
- Improved fault tolerance or zero downtime maintenance
- Simplified system administration
- Simplified capacity expansion
- Simplified system installation
- Support for legacy systems and applications
- Simplified system-level development
- Simplified system and application testing
- Security

  The benefits of virtualization directly propagate into cloud computing and have empowered it as well.

Downsides Of Virtualization

- Single point of failure problem
- Lower performance issue
- Difficulty in root cause analysis

  The positive impulse of virtualization prevails over the negatives by far

Virtualization Security Threats

- The single point host
- Threats to hypervisor
- Complex configuration
- Privilege escalation
- Inactive virtual machines
- Consolidation of different trust zones

  Any virtualization threats can be mitigated by maintaining security recommendations while designing a computing system

Virtualization Security Recommendations

- Hardening virtual machines
- Hardening the hypervisor
- Hardening the host operating system
- Restrictive physical access to the host
- Implementation of single primary function per vm
- Use of secured communications
- Use of separate nic for sensitive vm

Guest OS, hypervisor, host OS and the physical system are the four layers in the architecture of virtualized environment, and for security measures all of these should be considered separately.

Virtualization and Cloud computing (1/2)

- The resources at data center are virtualized and it is usually referred as data center virtualization
  - Pools of resources are created at data centers and a layer of abstraction is created over the pools of various types of physical resources using virtualization
  - Consumers of cloud services can only access the virtual computing resources from the data center
- Data center virtualization provides a way to the cloud system for managing resources efficiently

Data center virtualization is one foundation of cloud computing. Accesses to pooled resources is provided using resource virtualization.

Virtualization and cloud computing (2/2)

- Virtualization is the key enabler of most of the fundamental attributes of cloud computing
  - Shared service: Users remain unaware about the actual physical resources and cannot occupy any specific resource unit while not doing any productive work.
  - Elasticity: With virtualized resources, the underlying capacity of actual resources can be easily altered to meet the varying demand of computation
  - Service orientation: The implementation of service-oriented the architecture becomes easier when virtualization is in place.
  - Metered usage: In cloud computing, the services are billed on usage basis. The accurate measurement of resource consumption has been possible due to use of virtualized resources