











# Λογισμικό Συστήματος Systems Programming

Τμήμα ΗΜΜΥ

# Lesson Overview – Systems Programming

Deeper understanding in OS & systems programming

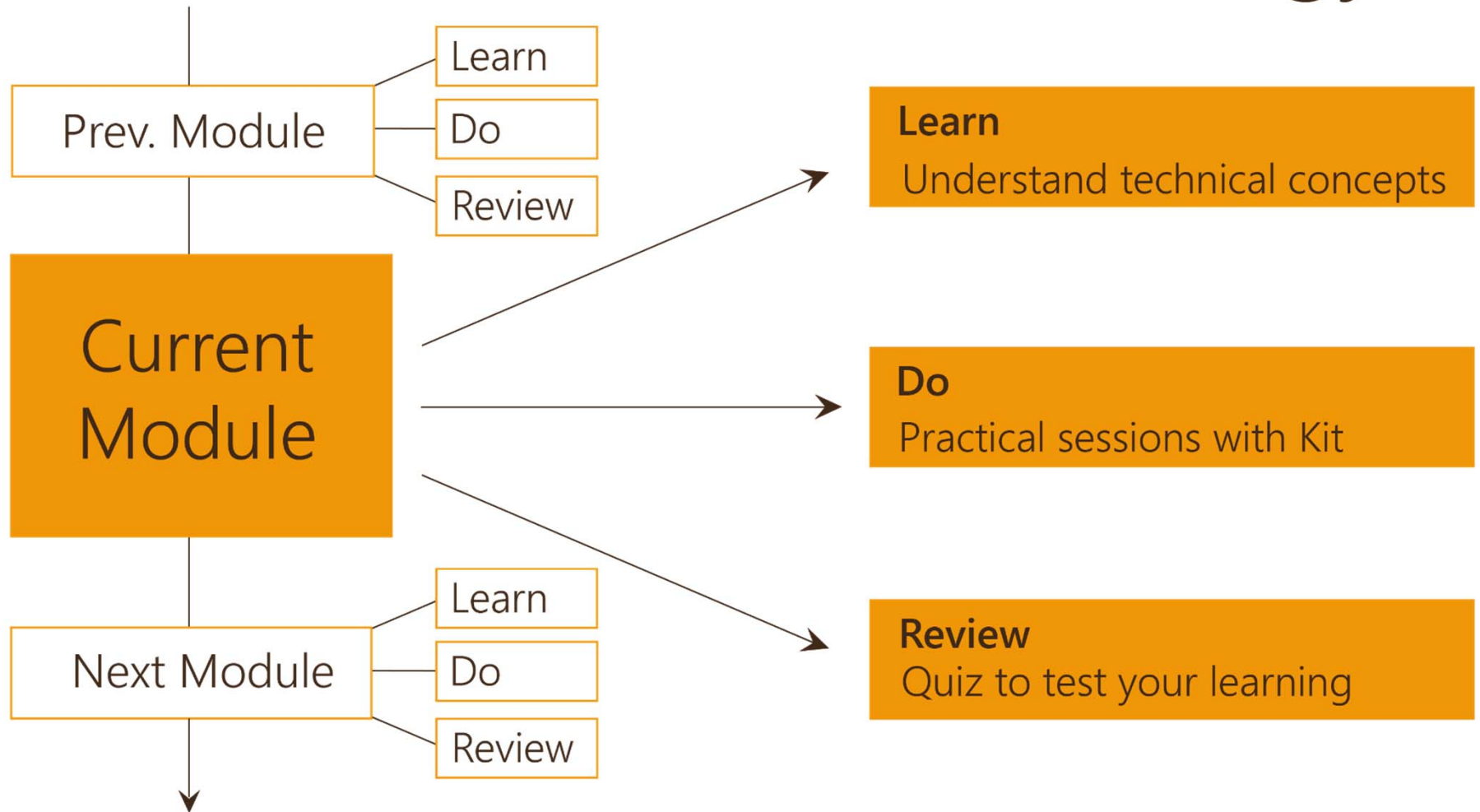
	x01_CLASS_SW_DEVEL_MAKE_GDB_LINKING.zip
	x01_LAB_SW_DEVEL_LINUX_TOOLS_MAKE_GDB_LINKING.zip
	x02_CLASS_PROCESSES_SYNC_SHMEM_SIGNALS_PIPES.zip
	x02_LAB_PROCESSES_SHMEM_SIGNALS_PIPES.zip
	x03_CLASS_PTHREADS.zip
	x03_LAB_PTHREADS.zip
	x04_CLASS_NETW_PROGR.zip
	x04_LAB_NETW_PROGR.zip
	x05_CLASS_CACHE_PERF_TOOLS_DEBUG.zip
	x05_LAB_CACHE_PERF_TOOLS_DEBUG.zip

**3 Projects (up to 50%)**

	x06_CLASS_ARDUINO_PT_SERIAL_IP.zip
	x06_LAB_ARDUINO_PT_SERIAL.zip
	x07_CLASS_ARDUINO_RTOS.zip
	x07_LAB_ARDUINO_RENESAS_RTOS.zip
	x08_CLASS_LDD.zip
	x08_LAB_LDD_1.zip
	x08_LAB_LDD_2.zip
	x08_LAB_LDD_3.zip
	x08_LAB_LDD_4.zip

basis for most other programming  
and efficient system use

# Learn-Do-Review Methodology



# Intellectual Property in HW & SW Packages

- ❑ **Patents** protect "inventions" that are useful, nonobvious and novel, and must be approved by an Office (e.g., USPTO)
  - ❑ A patent allows to exclude others from making, using, selling, and importing an invention for limited time, usually twenty years
- ❑ **Copyright** applies automatically to "works of authorship" in a tangible medium (paper, CD, book)
  - ❑ It prohibits reproduction, distribution, modification, public performance and display of software that is "substantially similar" to the original
  - ❑ All SW is automatically protected by copyright!
    - ❑ 80% on GitHub bear no license

# Intellectual Property in HW & SW Packages

- ❑ **Trade secret (not trademark)** protects info misappropriation
  - ❑ It prevents wrongful taking of a formula, practice, process, design, instrument, pattern, or commercial method
  - ❑ Trade secret law protects secrets in the source code of the software
  - ❑ Trade secret law protects the holder of a trade secret "against the disclosure or unauthorized use of the trade secret by those to whom the secret has been confided ...
  - ❑ Trade secret law does not protect a trade secret holder against discovery of that trade secret by "fair and honest means, such as by independent invention, accidental disclosure, or by so-called reverse engineering."<sup>+</sup>

www.opensource.org

www.fsf.org

**OPEN-SOURCE**

**FREE**

**SOFTWARE**

**ACCESS**

**COPYRIGHT**

**PUBLIC**

**CONTENT**

**INFORMATION**

**OPERATING**

**COMMONS**

**PROJECT**

**NETWORK**

**BUSINESS**

**CREATIVE**

**RESOURCES**

**MODEL**

**PRODUCTS**

**SHARING**

**USERS**

**PRODUCTION**

**ONE**

**CULTURAL**

**HARDWARE**

**INNOVATION**

**EXAMPLES**

**INTELLECTUAL**

**PROPERTY**

**GENERAL**

**CREATED**

**CODE**

**RESEARCH**

**AVAILABLE**

**FILE**

**DEVELOPMENT**

**SCIENCE**

**FIRST**

**CULTURE**

**USE**

**SYSTEM**

**ETHICS**

**LINK**

**INDIVIDUALS**

**SHARED**

**EXAMPLE**

**DESIGN**

**INCLUDING**

**SCIENTIFIC**

**WORLD**

**PROJECTS**

**POTENTIAL**

**COMMUNITIES**

**DISTRIBUTED**

**PRODUCT**

**MANUFACTURERS**

**SIMILAR**

**TECHNOLOGY**

**PLATINUM**

**ANOTHER**

**WORKS**

**POLITICAL**

**NEW**

**ARTICLE**

**MAKING**

**SOCKET**

**BUG**

**PROCESS**

**COMPANIES**

**WELL**

**COMMUNITY**

**TECHNOLOGIES**

**MANY**

**MADE**

**INTERNET**

**TERMS**

**SITE**


**DIGITAL**

**USING**

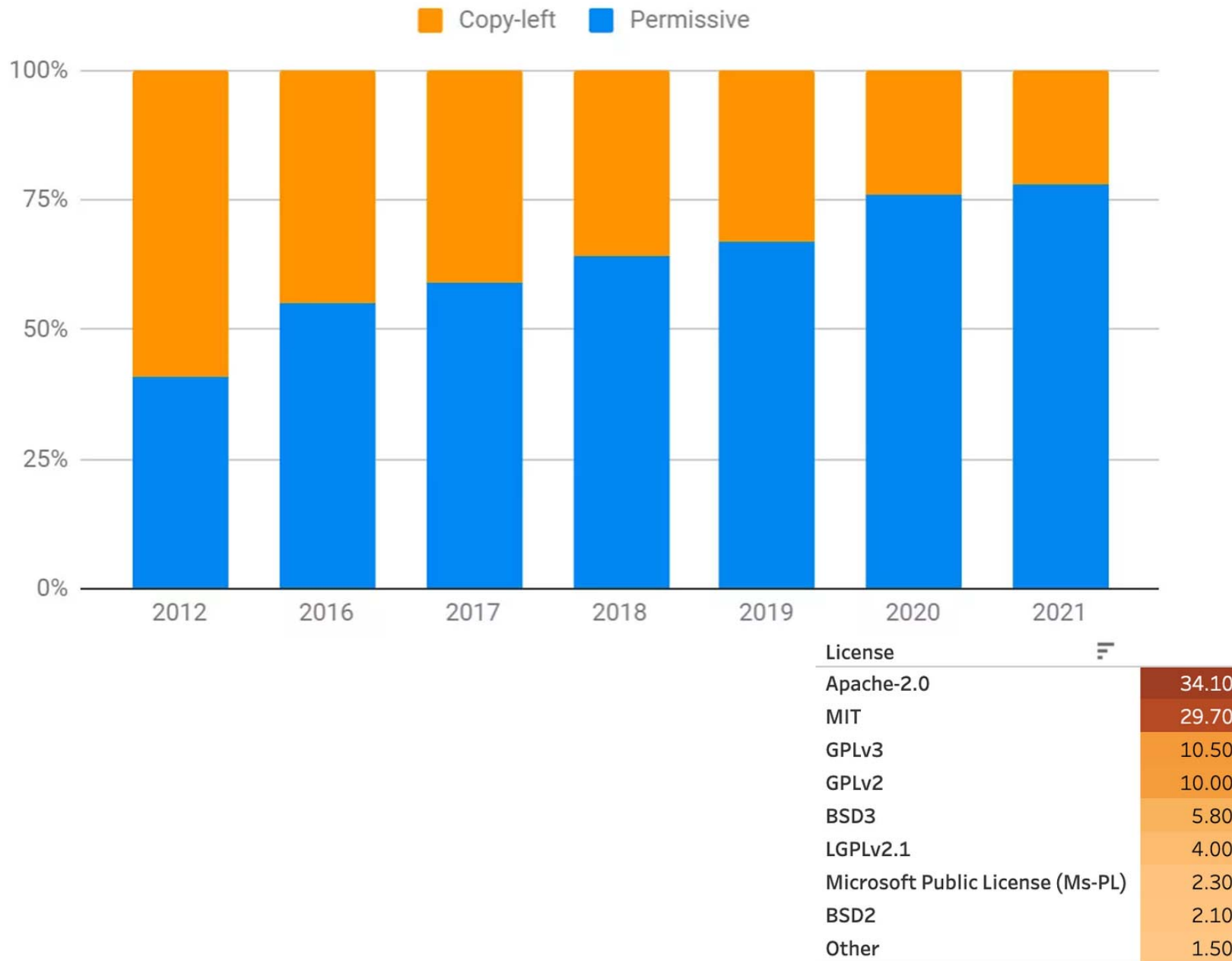
**ONLINE**

7

# IP Rights - Free & Open-Source SW

- ❑ **Open-source software** makes source code available for free via a license that concerns software
  - ❑ use
  - ❑ study
  - ❑ modification
  - ❑ distribution
- ❑ **Permissive sw license** (BSD-like) is a free software license with minimal requirements on how the software can be redistributed
  - ❑ Examples: MIT, BSD, Apple Public Source, and Apache
-  **Copyleft sw license** enforces freedom to distribute modified versions provided that the same rights be preserved in derivative works
  - ❑ Examples: GPL2, **GPL3**, LGPL, AGPL, ISC
  - ❑ Most famous one is the GPL2 license used with Linux
- ❑ Standish Group (from 2008) states that adoption of open-source software models has resulted in savings of about \$60 billion per year to consumers

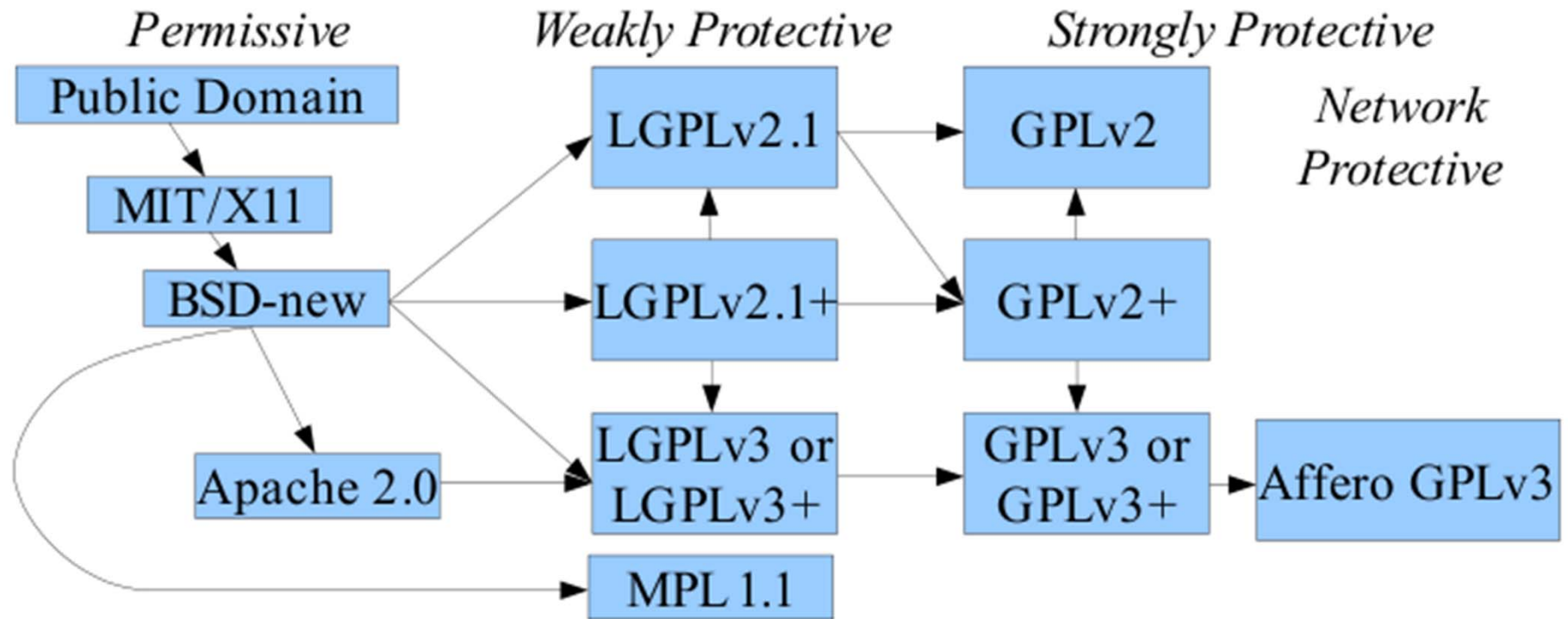
# IP Rights - Free & Open-Source SW



Top licenses looked like in 2021. Data: MEND



# License Compatibility



To see if software can be combined, start at their respective licenses and find a common box that can be reached by arrows from each license. Other possibilities exist if you are only using software as a library.

# Free & Open-Source SW

FSF: “The fundamental difference between the two movements is in their values, their ways of looking at the world. For the Open Source movement, the issue of whether software should be open source is a practical question, not an ethical one. As one person put it, “Open source is a development methodology; free software is a social movement.” For the Open Source movement, non-free software is a suboptimal solution. For the Free Software movement, non-free software is a social problem and free software is the solution. ”

# Views & Opinions

<https://www.fsf.org> & <https://sfconservancy.org>

- ❑ Stallman differentiated among three classes
  - ❑ Works of **practical use** should be free
  - ❑ **Points of view** should be shareable but not changeable and
  - ❑ Works of **art/entertainment** should be copyrighted (only for 10 years)
    - ❑ Video games as software should be free but not their artwork
- ❑ In 2015, Stallman also advocated for free hardware designs
  - ❑ Also, patent-left movement
  - ❑ [https://en.wikipedia.org/wiki/Open-source\\_hardware](https://en.wikipedia.org/wiki/Open-source_hardware)
  - ❑ <https://freedomdefined.org/OSHW>
- ❑ More on Licenses
  - ❑ <https://www.gnu.org/licenses>



# GNU/Linux

- ❑ Linux is a free open-source operating system kernel built originally by a student to replace UNIX
  - ❑ Stallman started to develop GNU and founded FSF
  - ❑ Linus started to develop a new Linux Kernel from Minix, targeting for commercial
- ❑ Linux started in early 1990s and is still under development
  - ❑ Linus team supports the kernel
  - ❑ GNU sponsors many tools for software dev (gnutils)
  - ❑ First choice for enterprise servers supercomputers (100%), embedded systems (50%). and geeks
  - ❑ rare on desktop (x86, x86/64, PPC), now ~5% (max ever)
  - ❑ server, cloud, TV, phone, watch, stereo, car info, thermostat, lock...



<https://www.levenez.com/unix>

<https://kernel.org>

<https://distrowatch.com>

<https://archiveos.org/linux>

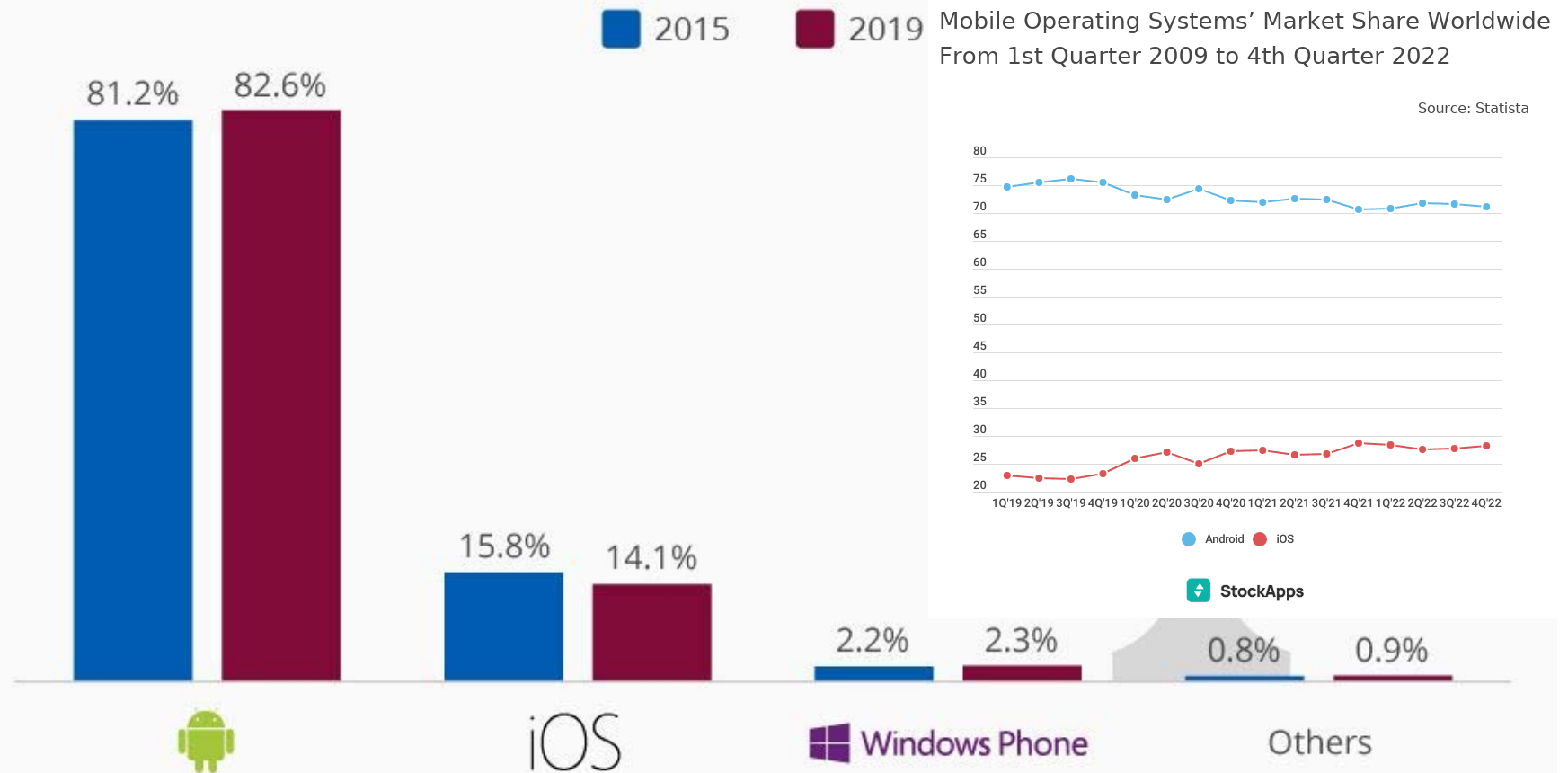
<http://lwn.net/Distributions>

It's fast, stable, uses less system resources and it gets out stains without causing

# Linux in Mobile Market

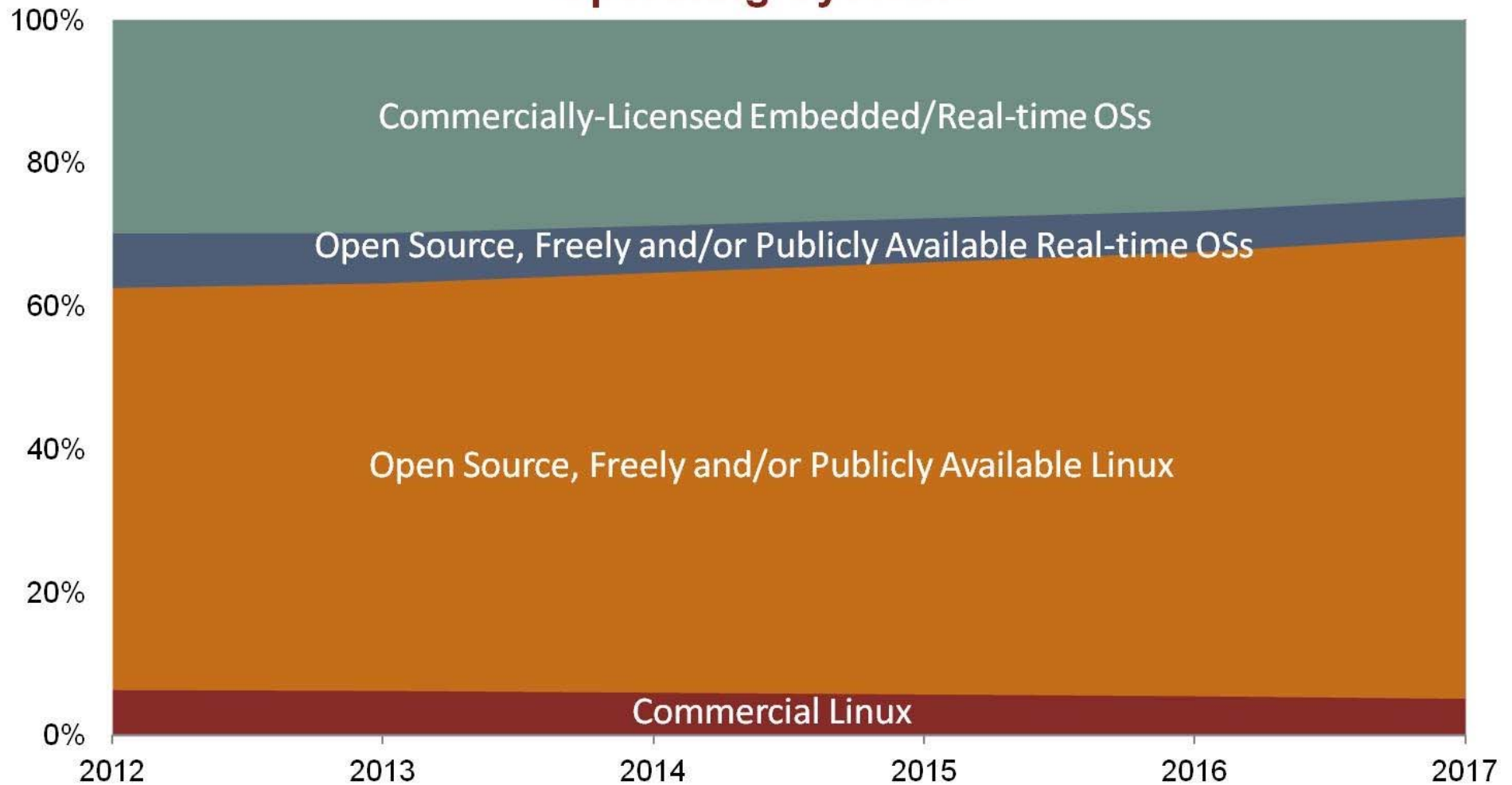
## The Platform War Is Over and Android Won

Worldwide smartphone operating system market share (% of new device shipments)\*



# Linux in Embedded Real-Time

## Worldwide Unit Shipments of Embedded/Real-time Operating Systems

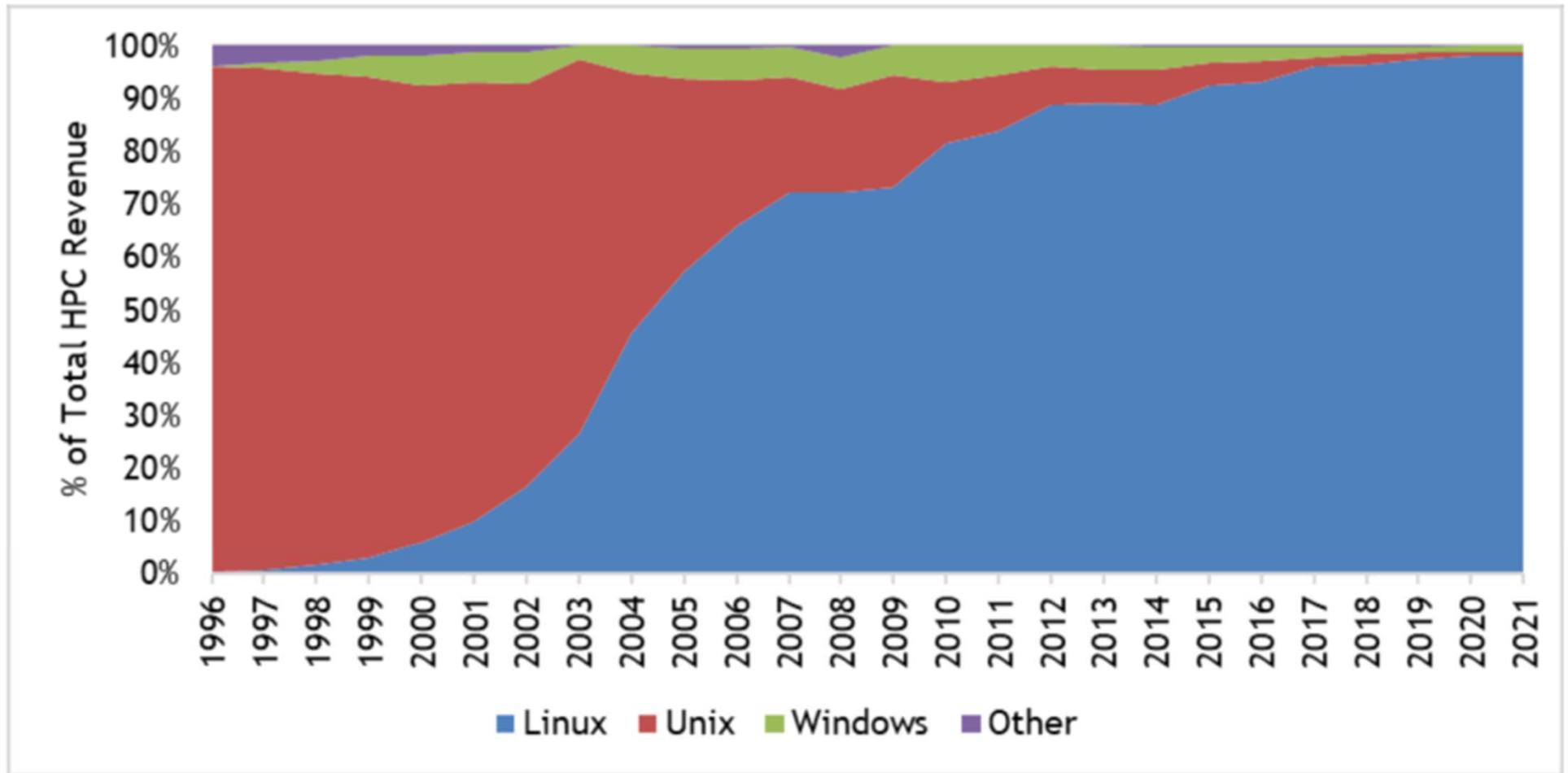


Note: More than one-third of embedded projects feature no formal OS or an in-house developed OS and are not depicted in the chart above. Source: VDC|Research, 2015



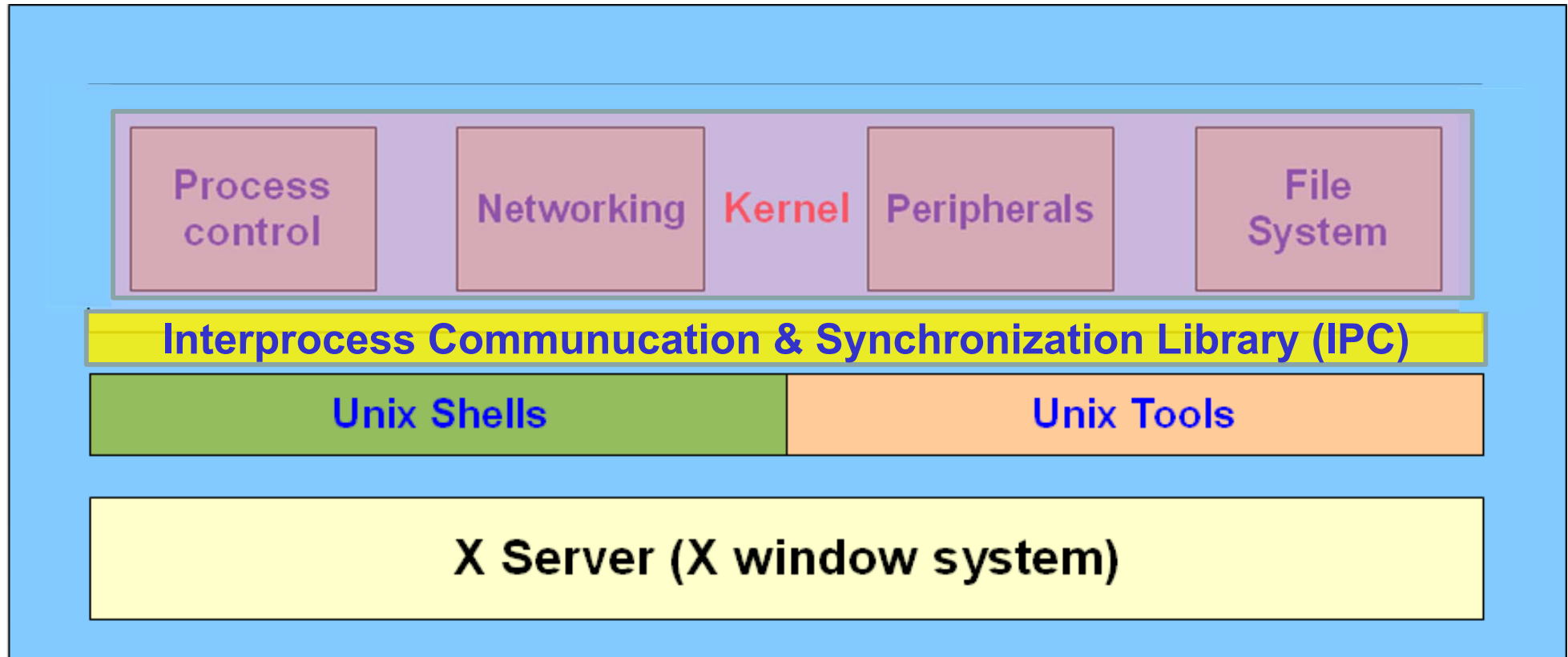
# Linux in HPC

Percent of HPC Server Revenue by Operating System



Source: Hyperion Research, 2022

# GNU/Linux Architecture



- ☐ Linux has a monolithic kernel
- ☐ GNU/Linux uses the structure of layered model
- ☐ Linux window managers (based on X11 and recently Waylan)
  - ☐ Examples: kde, Gnome, xfce, xpde, fluxbox, twm, NextStep, etc



# Linux Distros

<https://distrowatch.com>

- ☐ Linux is open source, so any one can develop a version
- ☐ Linux distros provide different default Window Managers & applications
- ☐ Standard distros from market: Redhat, Debian, Slackware
- ☐ Most distros based on Redhat (Fedora) or Debian (Ubuntu)
- ☐ Which one is suitable for me?
  - ☐ Redhat is good for enterprise work
  - ☐ Debian is good for professional end-users
  - ☐ Fedora is good for personal use and developing
  - ☐ Opensuse is good for training usage
  - ☐ Slackware & Arch put control/configuration in the hands of the user
  - ☐ CentOS is intended for professional users
  - ☐ Ubuntu is suitable for **ALL !!!**



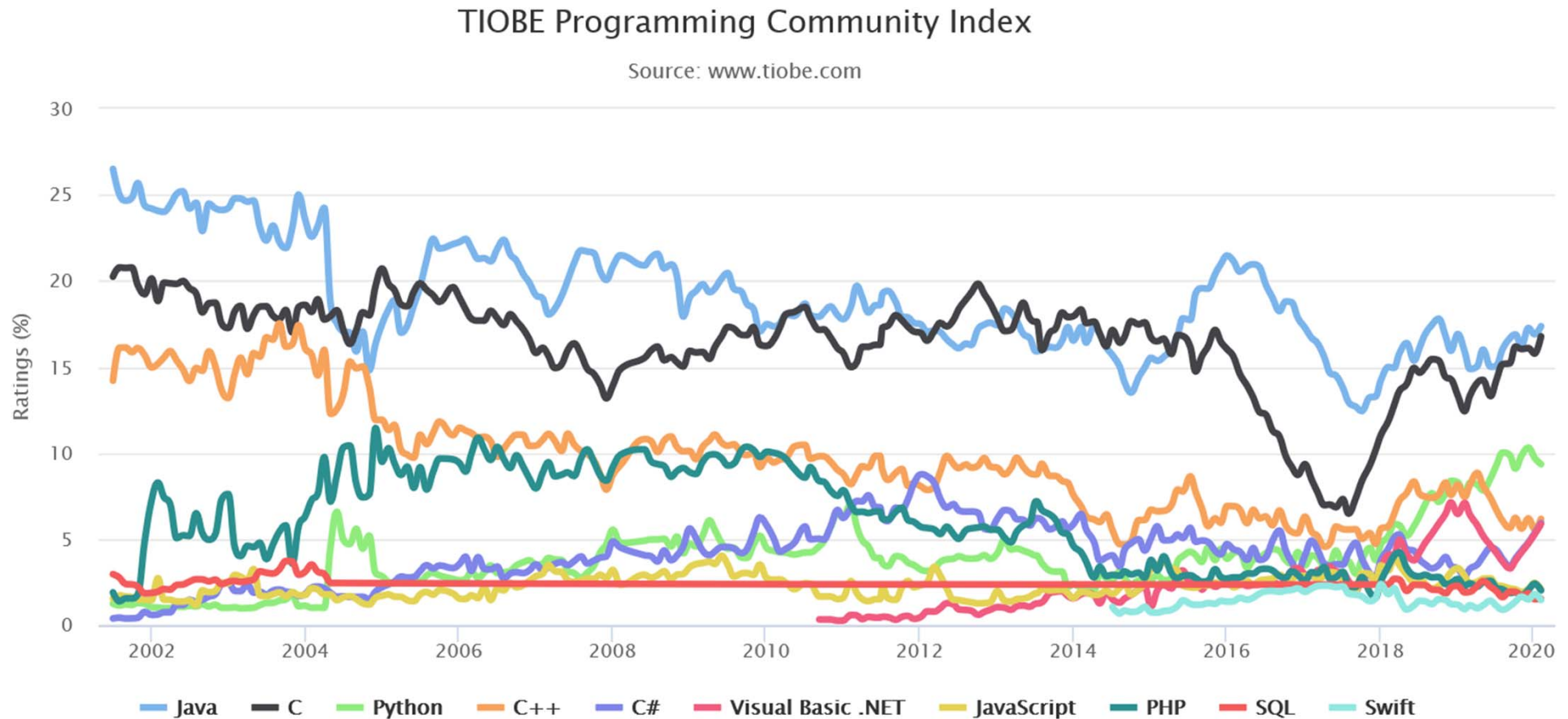
debian



slackware  
linux

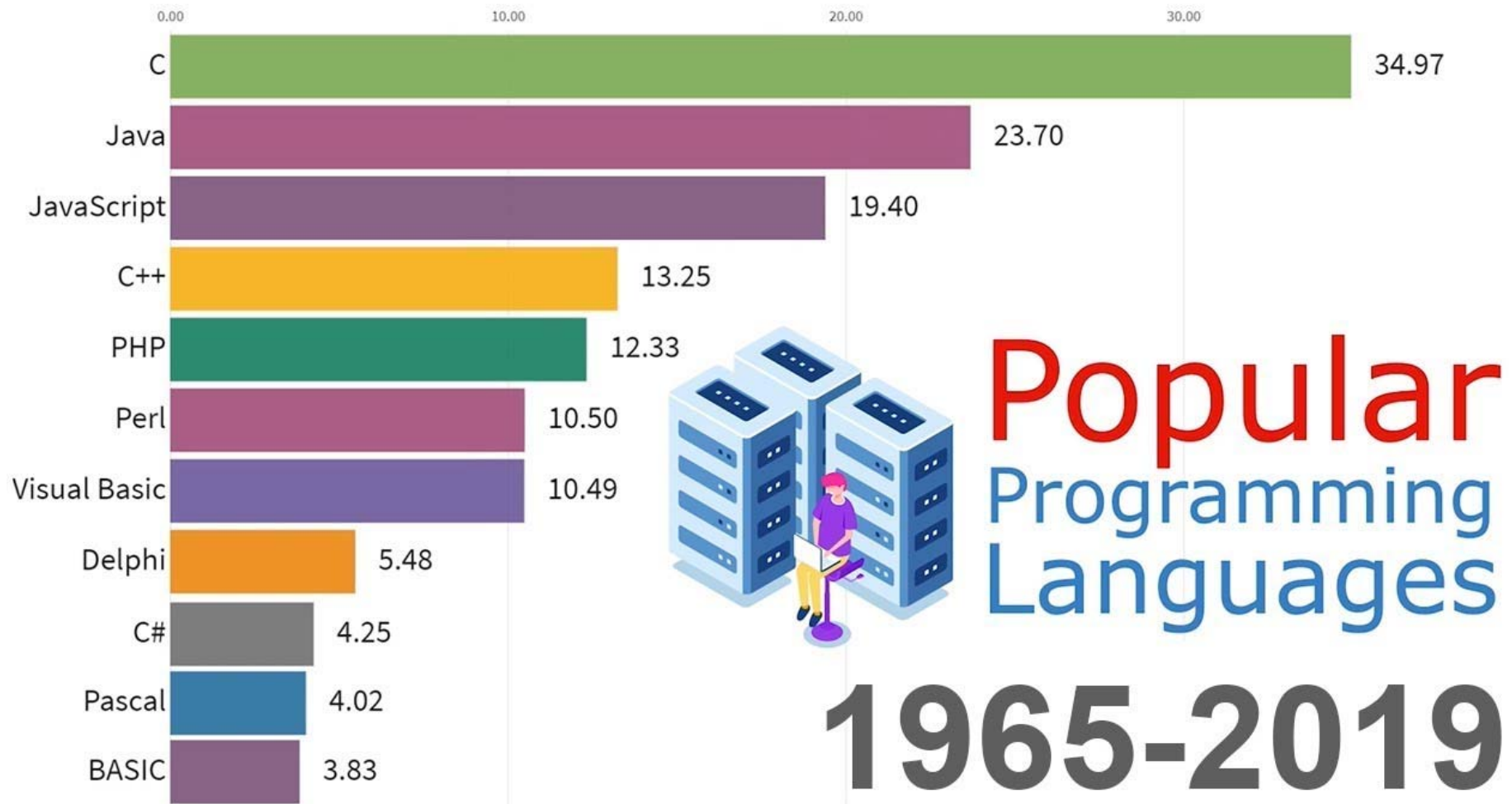


# Programming Languages - TIOBE Index



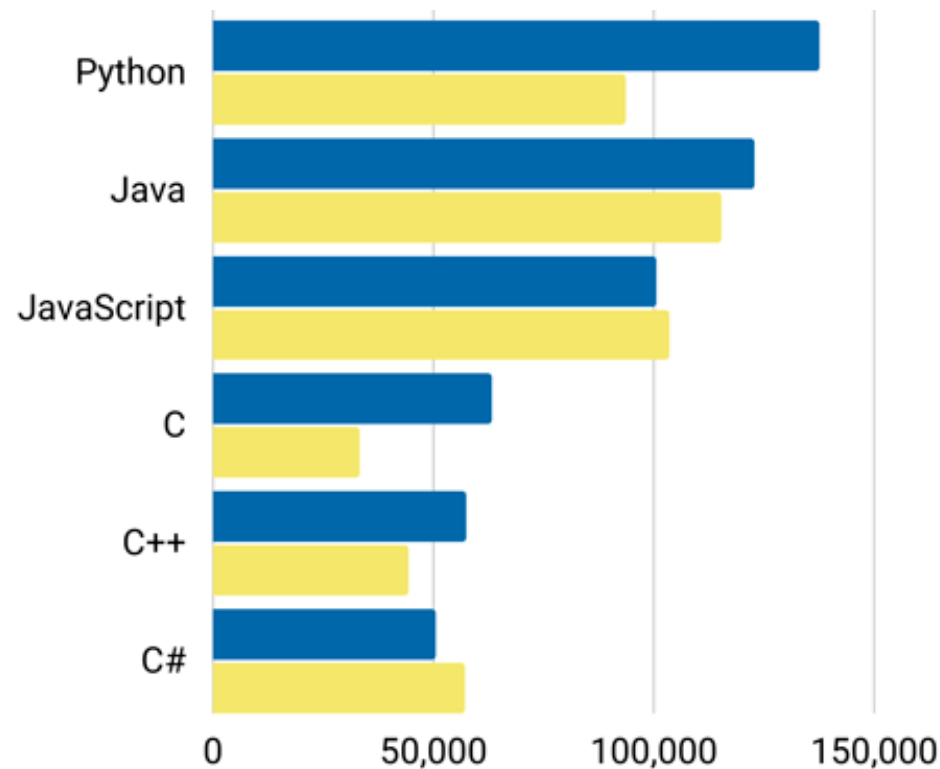
[index | TIOBE - The Software Quality Company](#)

# Programming Language Popularity



# Programming Language Jobs

## Most in-demand programming languages 2021-2022



■ US job posts ■ Europe job posts by: CodingNomads

# Languages in Embedded Systems

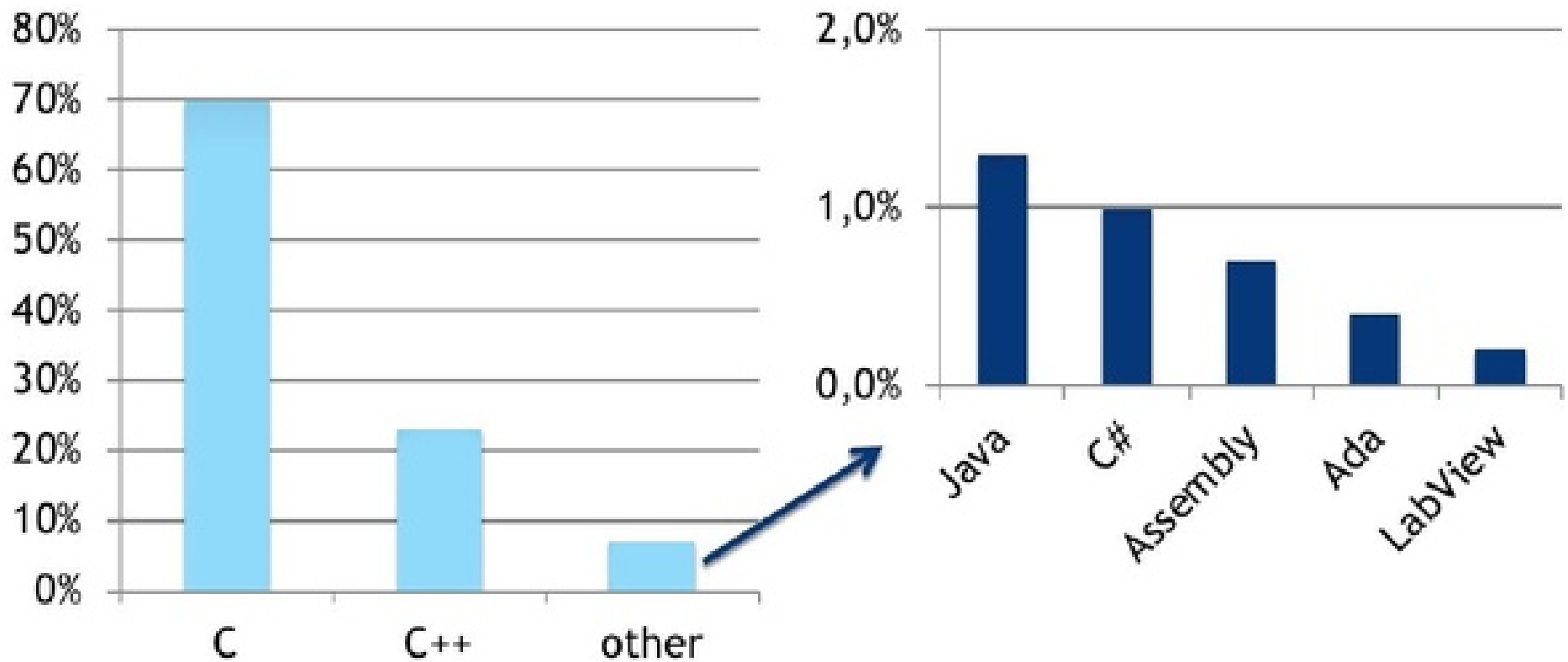


Figure 11. Primary Programming Language in Embedded Systems Designs

# GCC

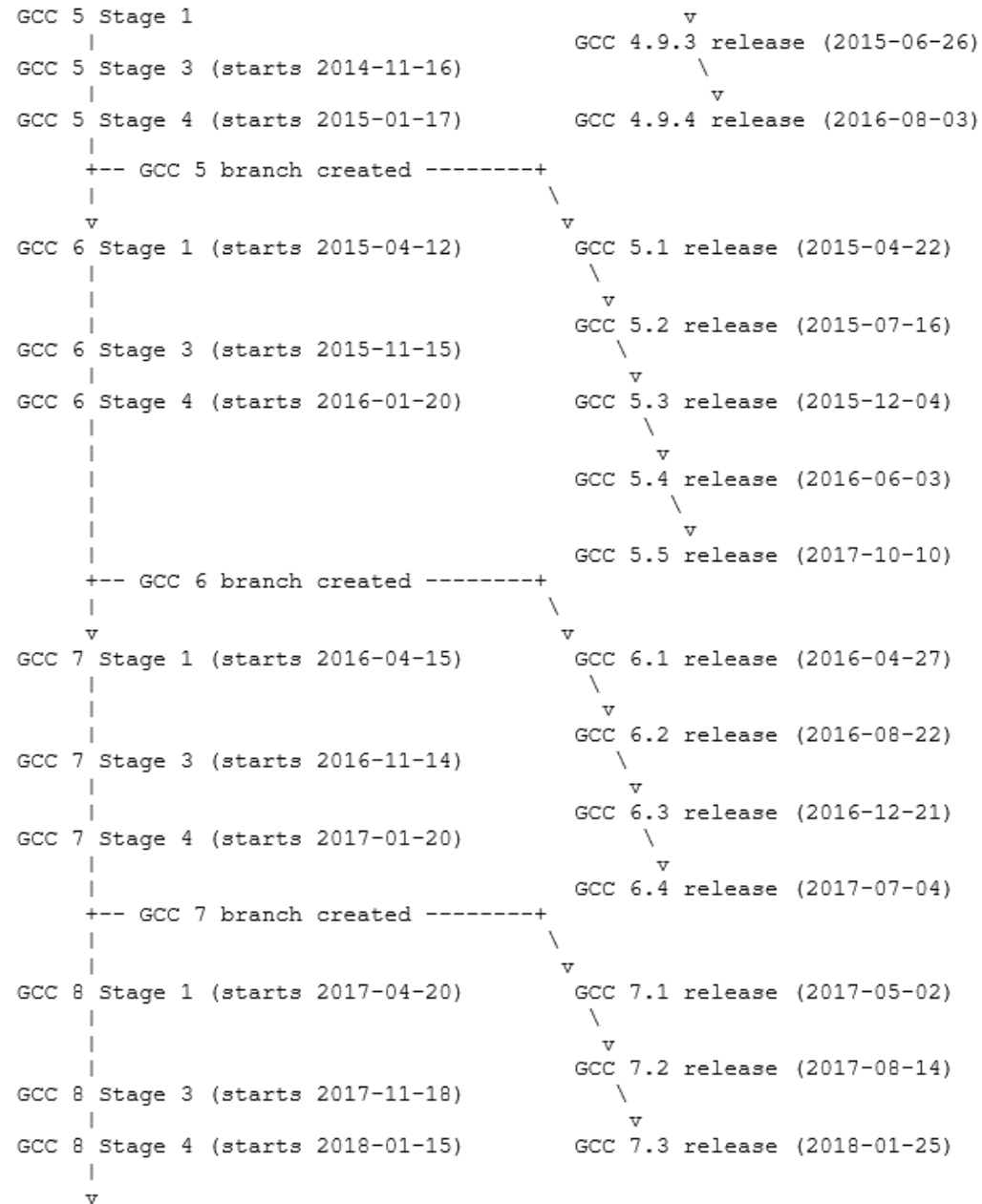
- ❑ **GNU Compiler Collection (GCC)** is a compiler system produced by the GNU Project
- ❑ GCC toolchain plays an important role in free SW
- ❑ GNU GPL
- ❑ GCC 1.0 in 1987
- ❑ Extended to C++ in same year (gcc for C, g++ for C++)
- ❑ Later front end for Objective-C, Objective-C++, Fortran, D, Ada, and Go
- ❑ How it works
  - ❑ The compiler front-end analyzes the source code to build an *intermediate representation* (lexical analysis, parsing)
  - ❑ It also manages the *symbol table*, a data structure mapping each symbol in the source code to associated information, such as address location, type and scope



# GCC

- ❑ GCC is primarily written in C
  - ❑ Complex version system (concurrent major, minor versions)
- ❑ In August 2012 (version 4.8+), the GCC steering committee announced that GCC now uses C++ as its implementation language. To build GCC from sources,
  - ❑ build new version of GCC with existing C compiler,
  - ❑ re-build new version of GCC with the one just built
  - ❑ repeat step 2 for verification purposes
- ❑ Optimization can occur during any phase of compilation
- ❑ Optimization depends on the release
  - ❑ Current version 13.2, see <https://gcc.gnu.org>

# GCC





# What The C Compiler Does

- ❑ The C compiler converts a C file from text into an *object file*
- ❑ On UNIX platforms these object files normally have a .o suffix; on Windows they have a .obj suffix
- ❑ The contents of an object file are
  - ❑ *code*, corresponding to **functions** (which may refer to data or other functions)
  - ❑ *data*, corresponding to **global** variables in the C file

# SW Development - Make



- ❑ **Make** developed at Bell Labs around 1978 by S. Feldman (now at IBM)
- ❑ *Makefile* **directs make** how to **compile & link** a program (based on dependencies)
  - ❑ When a C/C++ source file is **changed**, it must be **recompiled**
  - ❑ If a **header** file is **changed**, all C/C++ source files that include the header file must be **recompiled**
- ❑ Each compilation produces an **object** file corresponding to the source file
- ❑ All object files, whether new or old, are **linked** to produce the new executable
  - ❑ If none of the files has changed, **no actions** take place
- ❑ For **large** software projects, using Makefiles can substantially reduce build times since only few source files usually change.
- ❑ Therefore, make
  - ❑ sorts out dependency relations among files
  - ❑ avoids having to rebuild the entire project after modification of a single source file
  - ❑ performs selective rebuilds following a dependency graph
  - ❑ allows simplification of rules through macros and suffixes

# Make

- ❑ Make is a command generator and build utility (used for large projects)
- ❑ Linux supports different utilities, GNU make, cmake, imake **etc**
  - ❑ autoconf/automake, libtool, ac-tools, and jam, or scons tools can automatically generate Makefiles
- ❑ Windows supports a variation with its nmake utility
  - ❑ Unix like Makefiles can be used in Windows (Cygwin or Mingw environment)
- ❑ Makefiles contain
  - ❑ *directive* (an instruction for make to do something special while reading the Makefile such as reading another makefile)
  - ❑ *variable definition* (a line that specifies a text string value for a variable that can be substituted into the text later)
  - ❑ *explicit rule* (how/when to remake one or more files)
  - ❑ *implicit rule* (how/when to remake a class of files based on their names)
  - ❑ '#' for comments
- ❑ Rule example

```
target: dependencies
      system command(s)
```

# Simple Makefile

- ❑ A *makefile* is executed with the `make` command (or `make -f`)
- ❑ `make clean` deletes executable file and all object files from the directory
- ❑ Simple Example

*describes how an executable file  
“edit” depends on four object files  
which, in turn, depend on four C  
source and two header files  
(cc can be replaced with gcc)*

```
edit : main.o kbd.o command.o display.o
      cc -o edit main.o kbd.o command.o display.o

main.o : main.c defs.h
      cc -c main.c
kbd.o : kbd.c defs.h command.h
      cc -c kbd.c
command.o : command.c defs.h command.h
      cc -c command.c
display.o : display.c defs.h
      cc -c display.c

clean :
      rm edit main.o kbd.o command.o display.o
```

# Another Makefile

```
CC = g++
CFLAGS = -g -Wall -DDEBUG -I/usr/home/neal/include
LDFLAGS = -L/usr/home/neal/lib -lgraph

PROG = example
OBJS = main.o binky.o akbar.o
HDRS = binky.h akbar.h defs.h
SRCS = main.c binky.cc akbar.cc

main.o : binky.h akbar.h defs.h parse.c
        $(CC) $(CFLAGS) -c main.c parse.c
binky.o : binky.h
akbar.o : akbar.h defs.h

$(PROG) : $(OBJS)
$(CC) -o $(PROG) $(LDFLAGS) $(OBJS)

TAGS : $(SRCS) $(HDRS)
etags -t $(SRCS) $(HDRS)

clean :
rm -f core $(PROG) $(OBJS)
```



Default  
Compilation  
(can be skipped)

# Installing Software From Tarballs

- ☐ `tar xzf <usually-gzipped-tar-file>`
- ☐ `cd <dir>`
- ☐ Possibly run `autoconf` (via `.sh` script)
- ☐ `./configure` (finds system tools and generates Makefiles from `Makefile.am` skeletons, `config.status`, `log`, C header)
- ☐ `make` (or `make -f Makefile target`)
- ☐ `make check`
- ☐ `make install`
- ☐ `make clean/distclean` (cleanup partly or even Makefiles)
  
- ☐ Cross-platform software development?
- ☐ Autotools {`autoconf` creates `configure`, `automake` used in `configure`, `autotest`, ...}

<http://www.gnu.org/manual>

[GNU Autoconf, Automake and Libtool](http://www.gnu.org/manual)  
([sourceware.org](http://sourceware.org))

<http://autotoolset.sourceforge.net>

<http://autotoolset.sourceforge.net/tutorial.html>

# Alternative: Package Management

- ❑ Linux addon software is installed via Package Managers
- ❑ A package contains files & instructions to setup software
- ❑ Many packages depend on each other
- ❑ High-level managers download packages, figure out dependencies and deal with groups of packages
- ❑ Low-level managers unpack individual packages, run scripts, and get the software installed correctly
- ❑ Two main families exist:
  - ❑ `dpkg` and `rpm` (low level Debian or Redhat package managers)
  - ❑ `apt-get` (Debian), `zypper` (SUSE), `yum` (Fedora) at high level

# Debian (Advanced Package Tool, APT)

- ❑ `sudo dpkg -l //list installed packages`
- ❑ `sudo apt search myprogname // search for a package`
- ❑ `sudo apt install myprogname // install a package`
- ❑ `sudo apt update // update installed packages`
- ❑ `sudo apt remove myprogname // remove a package`
- ❑ `sudo apt policy myprogname //check package status`



# OpenSuse (Zypper)

- ❑ `sudo rpm -qa // list installed packages`
- ❑ `sudo zypper search myprogname // search for a package`
- ❑ `sudo zypper install myprogname // install a package`
- ❑ `sudo zypper update // update installed packages`
- ❑ `sudo zypper remove myprogname // remove a package`
- ❑ `sudo zypper info myprogname //check package status`

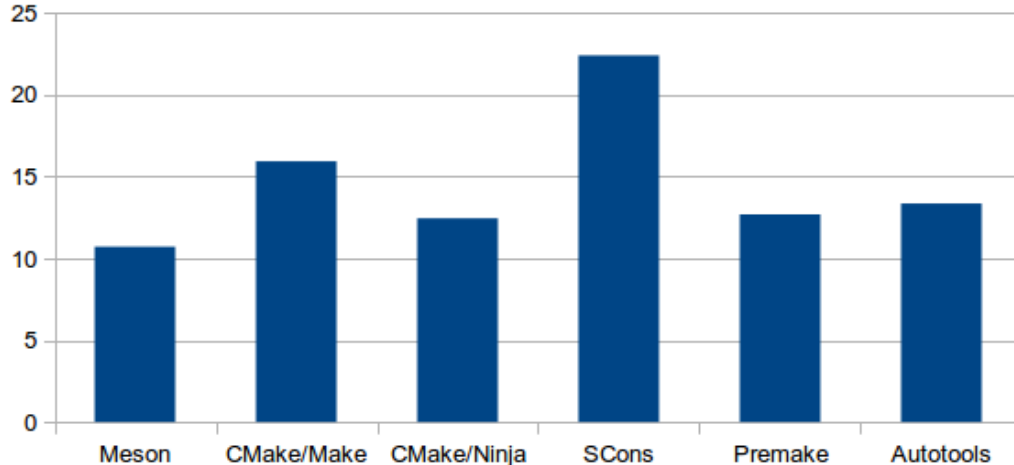
# Fedora/CentOS (Yellowdog Update Mod, YUM)

- ❑ `sudo yum list installed` // list installed packages
- ❑ `sudo yum search myprogname` // search for a package
- ❑ `sudo yum install myprogname` // install a package
- ❑ `sudo yum update` // update installed packages
- ❑ `sudo yum remove myprogname` // remove a package
- ❑ `sudo yum info myprogname` //check package status

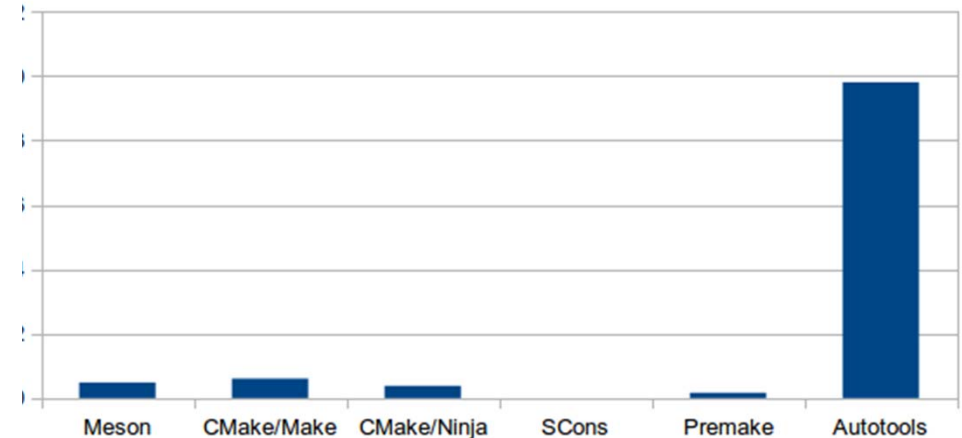
# Automatically Generate Makefiles

- ❑ Autotools (autoconf, automake, etc) & GNU Make
- ❑ Other ways
  - ❑ CMAKE & Make
  - ❑ CMAKE & Ninja
  - ❑ MESON
  - ❑ SCONS

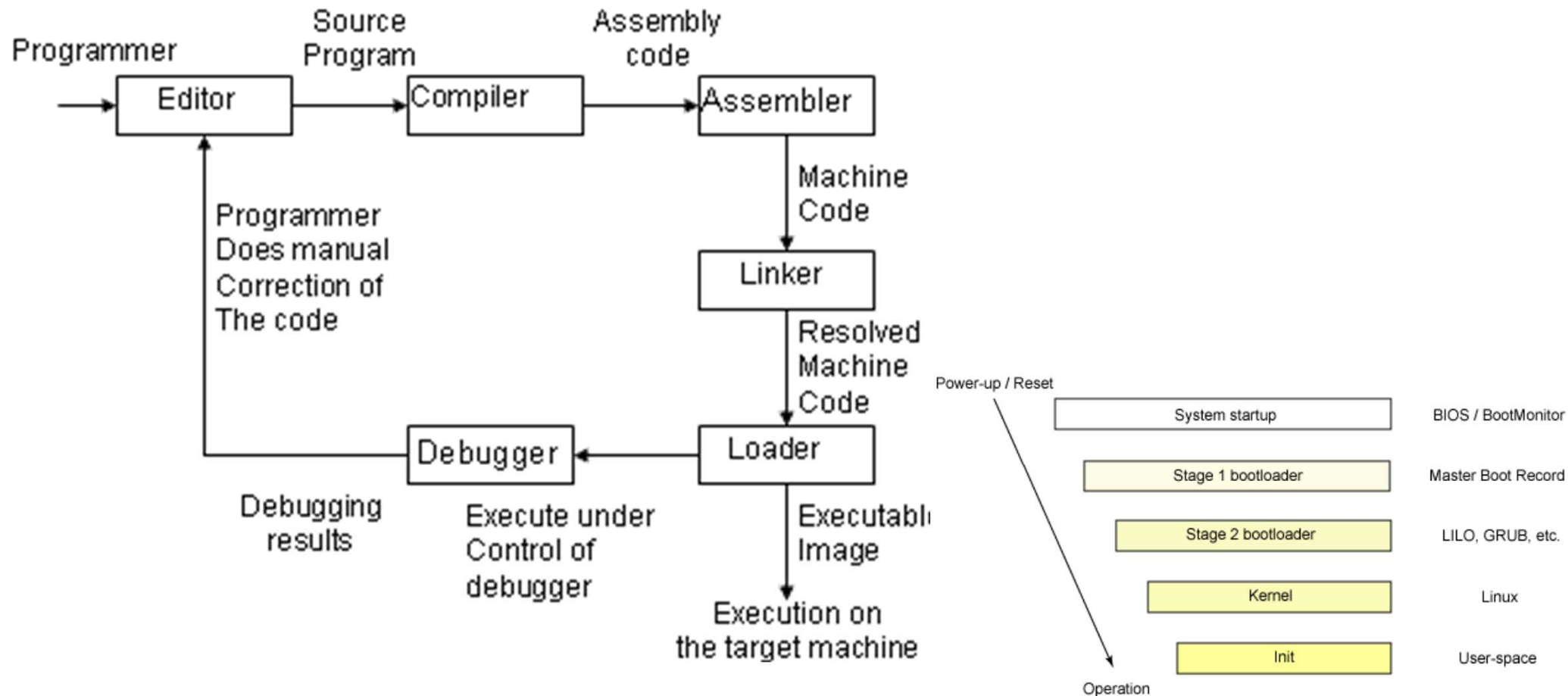
Build time



Configuration time



# System Software Tools



# Editor

- ❑ Core functionality
  - ❑ syntax highlighting
  - ❑ efficient keyboard maneuvering
  - ❑ setting markers, using buffers
  - ❑ copy, yank, fold, e.g., blocks
  - ❑ search and replace
  - ❑ window splitting
  - ❑ autocompletion
  - ❑ jump to definition / manual page
  - ❑ applying external commands and filters
  
- ❑ Powerful editors: vim or emacs?

# Compiler Toolchain

- ❑ A compiler translates source code from a high-level programming language into machine code for a given architecture. It performs a number of steps:
  - ❑ lexical analysis
  - ❑ preprocessing
  - ❑ parsing
  - ❑ semantic analysis
  - ❑ Assembly
  - ❑ code optimization
  - ❑ code generation
  - ❑ linking

# Compiler Toolchains

## ❑ Closed- and Open-Source Compilers

- ❑ Intel C/C++ Compiler (`cc` or `icc`)
- ❑ Turbo C / Turbo C++ / C++Builder (Borland)
- ❑ Microsoft Visual C++
- ❑ ...
- ❑ Clang (a frontend to LLVM)
- ❑ GNU Compiler Collection (`gcc`)
- ❑ Portable C Compiler (`pcc`)
- ❑ ...

## ❑ The compiler chain usually performs preprocessing (via `cpp(1)`), compilation (`cc(1)`), assembly (`as(1)`) and linking (`ld(1)`)

## ❑ Example

- ❑ `gcc -v -O3 -DFOOD=\"Avocado\" -E hello.c -o hello.out`  
(can disassemble via `objdump`, or `diff -bu hello_old.out hello.out`)

# Debug - GDB

- ❑ The **GNU debugger** `gdb` is the standard debugger for Linux (all distros)
- ❑ It is **portable** and runs on many Unix-like systems and languages, including Ada, C, C++, Objective-C, Free Pascal, Fortran, Java and others
- ❑ GDB offers extensive facilities for **tracing** or **altering** execution of programs
- ❑ User can **monitor/modify values** of internal program variables and/or **call functions** independent of program behavior
- ❑ GDB offers a 'remote' mode when debugging embedded systems
- ❑ GDB does not contain its own graphical user interface
  - ❑ Several front-ends, e.g., DDD



# Simple Example with GDB

- ❑ How to compile?

-ggdb

```
$ gcc example.c -g -o example
```

- ❑ How to Run

```
$ gdb ./example
```

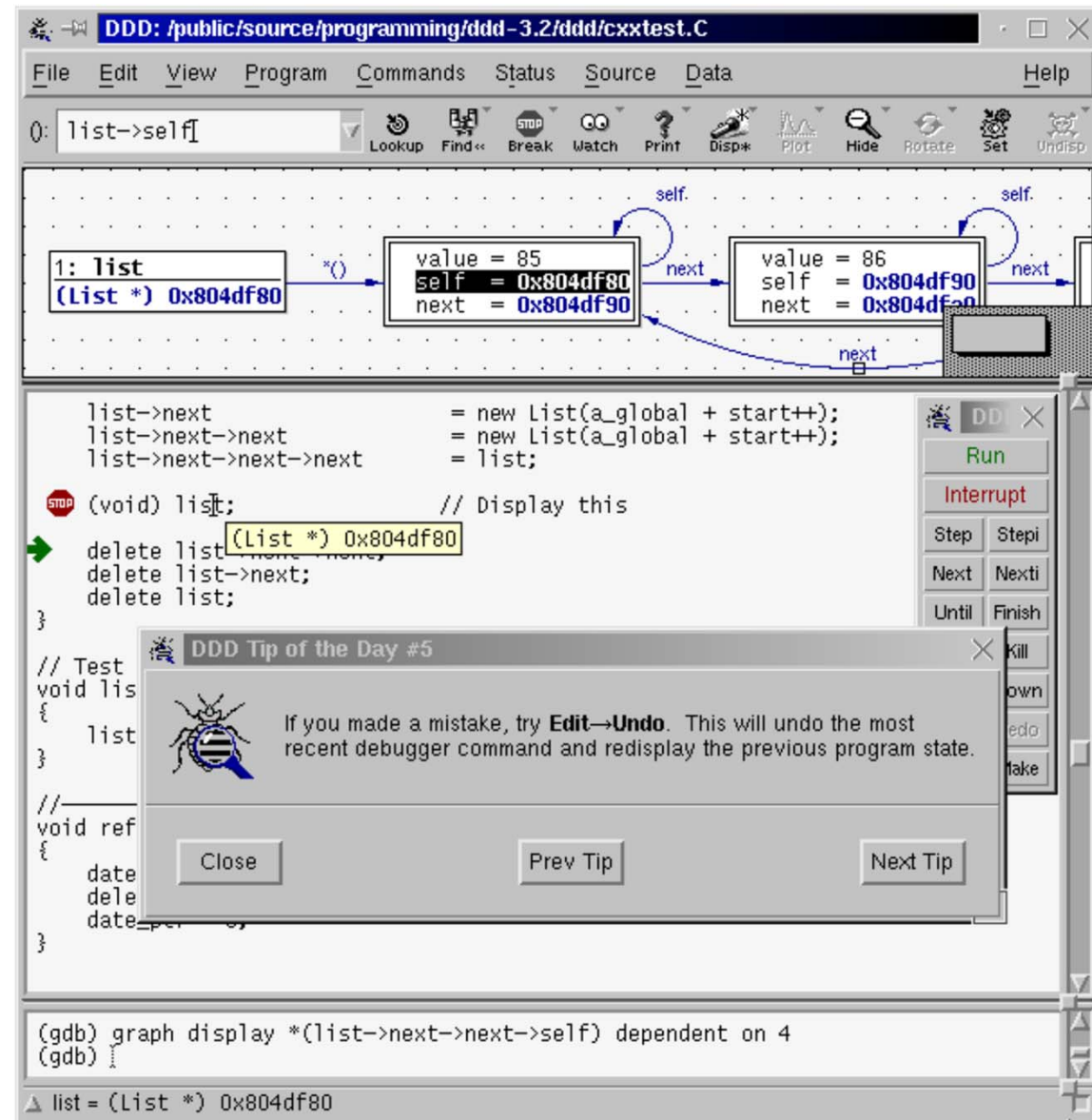
- ❑ Examples of commands

<b><code>gdb program</code></b>	debug "program" (from the shell)
<b><code>run -v</code></b>	run the loaded program with the parameters
<b><code>bt</code></b>	backtrace (in case the program crashed)
<b><code>info registers</code></b>	list registers in use
<b><code>q</code></b>	quit
<b><code>watch condition</code></b>	Suspend processing when condition is met. i.e. <code>x &gt; 5</code>
<b><code>break line-number</code></b>	Suspend program at specified line number

# DDD: Graphical GDB Frontend

- Allows to navigate in the source code while debugging
- Allows to set break points, inspect and change variable value directly by looking at the source code

[DDD - Data Display Debugger - GNU Project - Free Software Foundation \(FSF\)](#)



# Why Assembly (x86)

Understand machine language execution

- ☐ Understanding bugs, using debugger
- ☐ Optimizing the code
- ☐ Creating and fighting malware

Learning assembly is helpful for

- ☐ Device drivers
- ☐ OS kernel
- ☐ Game programming

# Why Assembly (x86)

31	0	
eax		Mostly general- purpose registers
ebx		
ecx		
edx		
esi		Source index
edi		Destination index
ebp		Base pointer
esp		Stack pointer
eflags		Status word
eip		Instruction Pointer (PC)

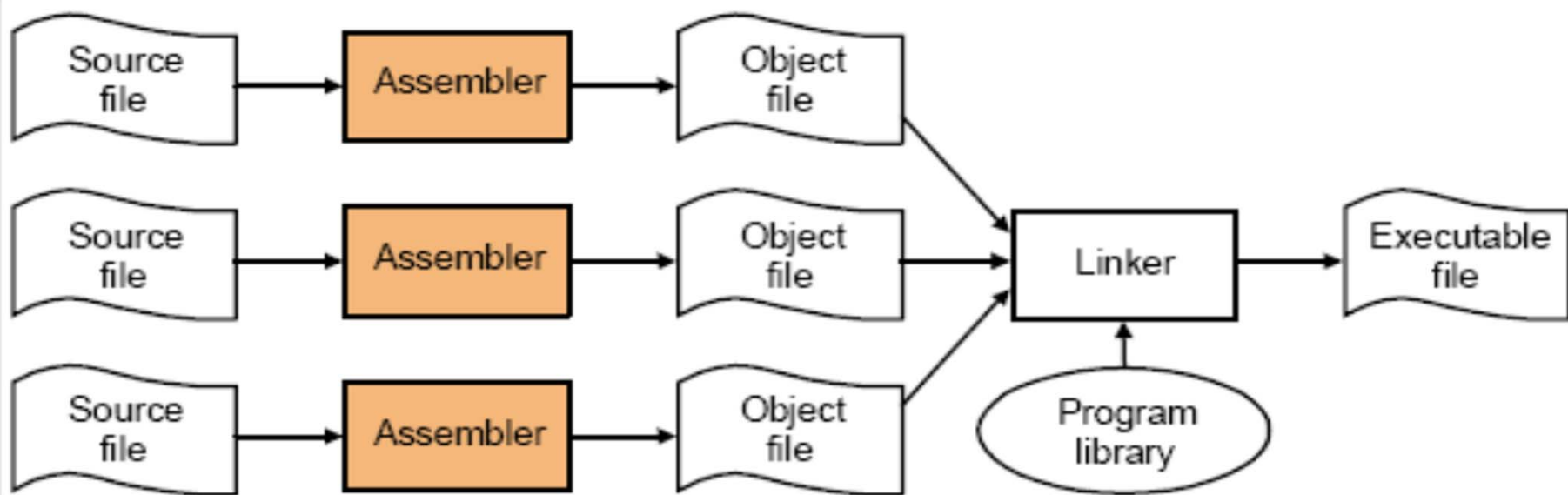
15	0	
cs		Code
ds		Data
ss		Stack
es		Extra
fs		Data
gs		Data

Segment  
Registers:

Added during  
address  
computation

- ☐ **Data movement**
  - ☐ mov, push, pop
- ☐ **Arithmetic/logic**
  - ☐ shift, add, sub
- ☐ **Port I/O**
  - ☐ in, out
- ☐ **Control jumps**
  - ☐ jz, call, ret
- ☐ **String**
  - ☐ rep movsl
- ☐ **System**
  - ☐ iret, int

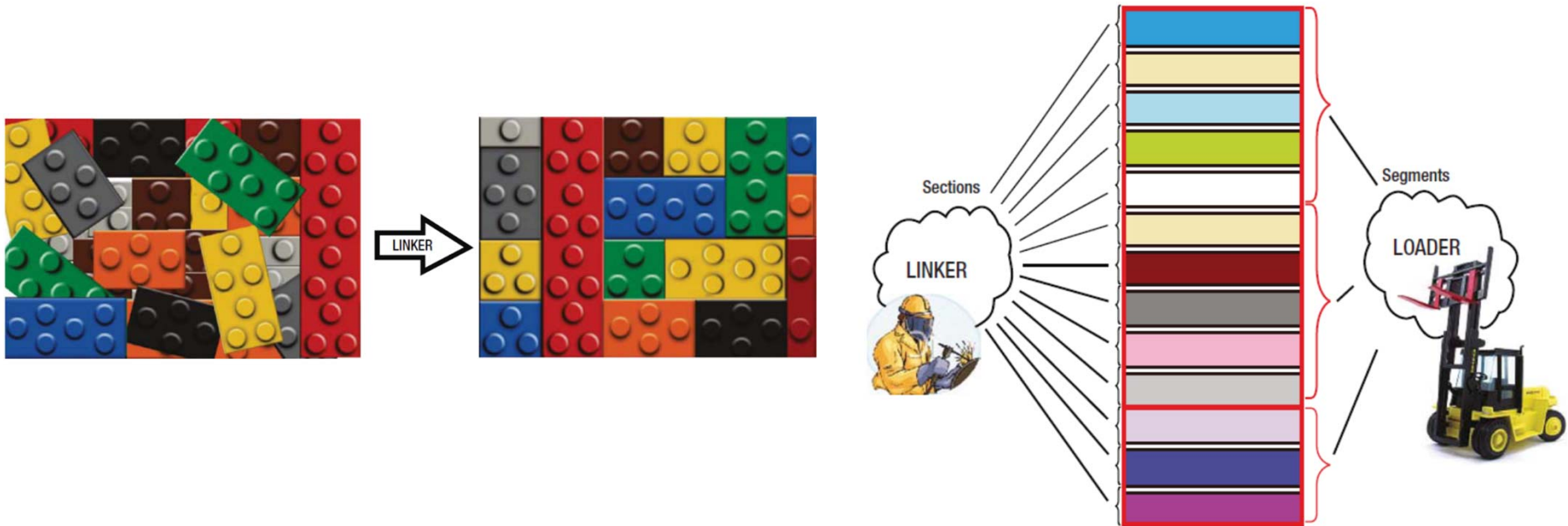
# Software – Compiler/Assembler & Linker



- ❑ A “*Linker*” accounts for and reconciles all address references within and among modules
  - ❑ Resolves inter-segment symbolic references
  - ❑ Determines and substitutes relocation of symbols attributes
  - ❑ Establishes correspondence between external and internal definitions

<https://lwn.net/Articles/276782/>

# Basic Functions of a Linker/Loader



- ❑ To execute an object program, we need
  - ❑ **Linking:** resolve symbolic reference in two or more separate object programs
  - ❑ **Relocation:** adjust addresses in object code so that program can be loaded at diff address
  - ❑ **Allocation & Loading:** Physically place code & data into memory and initiate execution
- ❑ Type of loaders
  - ❑ **assemble-and-go** loader
  - ❑ **absolute loader** (bootstrap loader)
  - ❑ **direct linking loader**
  - ❑ **relocating loader** (relative loader)



# Linker

- ❑ Program code written by one or more programmers
  - ❑ Compilers and translators translate one procedure at a time to **object code**
  - ❑ Linker works after **object code** is generated to
    - ❑ connect logically all these routines, and
    - ❑ link them a single executable
- ❑ A linker accounts for and reconciles all address references within and among modules and replaces references with a consistent scheme of relative addresses in an **executable binary program**
  - ❑ In **MS-DOS, Windows 95/98** etc object modules use extension **obj** and the executable binary programs use **exe** extension
  - ❑ In **UNIX**, object modules have **.o** extension and executable programs have **no extension** (default **a.out**)

# Object Code - ELF Format

## ■ Elf header

- Word size, byte ordering, file type (.o, exec, .so), machine type, etc.

## ■ Segment header table

- Page size, virtual addresses memory segments (sections), segment sizes.

## ■ .text section

- Code

## ■ .rodata section

- Read only data: jump tables, ...

## ■ .data section

- Initialized global variables

## ■ .bss section

- Uninitialized global variables
- "Block Started by Symbol"
- "Better Save Space"
- Has section header but occupies no space

ELF header
Segment header table (required for executables)
.text section
.rodata section
.data section
.bss section
.symtab section
.rel.txt section
.rel.data section
.debug section
Section header table



# Object Code - ELF Format

- **.symtab section**
  - Symbol table
  - Procedure and static variable names
  - Section names and locations
- **.rel.text section**
  - Relocation info for .text section
  - Addresses of instructions that will need to be modified in the executable
  - Instructions for modifying.
- **.rel.data section**
  - Relocation info for .data section
  - Addresses of pointer data that will need to be modified in the merged executable
- **.debug section**
  - Info for symbolic debugging (gcc -g)
- **Section header table**
  - Offsets and sizes of each section

ELF header
Segment header table (required for executables)
.text section
.rodata section
.data section
.bss section
.symtab section
.rel.txt section
.rel.data section
.debug section
Section header table

# Linker

- ❑ Variable **definition** makes the compiler reserve space and possibly fill that space with a particular value
- ❑ Function **definition** induces the compiler to generate code
- ❑ Definitions split into
  - ❑ *static global variables/functions* defined and referenced exclusively by a module
  - ❑ *non-static global variables/functions* defined by a module and referenced by other modules
  - ❑ *external global variables* referenced by a module but defined as globals by another module
- ❑ A **declaration** tells the C compiler that a definition exists elsewhere in the program, probably in a different C file

# Strong and Weak Symbols - Rules

**1.Relocation:** The linker takes the code and data sections of each object file and merges them into the code and data sections of the final executable file.

**2.Symbol Resolution:** The linker associates every symbol in the program with precisely one symbol definition.

- ❑ Program symbols are either strong or weak
  - ❑ Strong symbols: procedures and initialized globals
  - ❑ Weak symbols: uninitialized globals
- ❑ Linker Symbol Rules
  - ❑ Rule 1: Multiple strong symbols
    - ❑ Each strong item can be defined only once (else linker error)
  - ❑ Rule 2: Strong and multiple weak symbols
    - ❑ References to the weak symbol resolve to the strong symbol
  - ❑ Rule 3: If there are multiple weak symbols
    - ❑ Pick an arbitrary one
      - ❑ Can override this with `gcc -fno-common`

# Linker Code Example

```
/* This is the definition of a uninitialized global variable */
int x_global_uninit;

/* This is the definition of a initialized global variable */
int x_global_init = 1;

/* This is the definition of a uninitialized global variable, albeit
 * one that can only be accessed by name in this C file */
static int y_global_uninit;

/* This is the definition of a initialized global variable, albeit
 * one that can only be accessed by name in this C file */
static int y_global_init = 2;

/* This is a declaration of a global variable that exists somewhere
 * else in the program */
extern int z_global;

/* This is a declaration of a function that exists somewhere else in
 * the program (you can add "extern" beforehand if you like, but it's
 * not needed) */
int fn_a(int x, int y);
```

# Linker Code Example

```
/* This is a definition of a function, but because it is marked as
 * static, it can only be referred to by name in this C file alone */
static int fn_b(int x)
{
    return x+1;
}

/* This is a definition of a function. */
/* The function parameter counts as a local variable */
int fn_c(int x_local)
{
    /* This is the definition of an uninitialized local variable */
    int y_local_uninit;
    /* This is the definition of an initialized local variable */
    int y_local_init = 3;

    /* Code that refers to local and global variables and other
     * functions by name */
    x_global_uninit = fn_a(x_local, x_global_init);
    y_local_uninit = fn_a(x_local, y_local_init);
    y_local_uninit += fn_b(z_global);
    return (y_global_uninit + y_local_uninit);
}
```

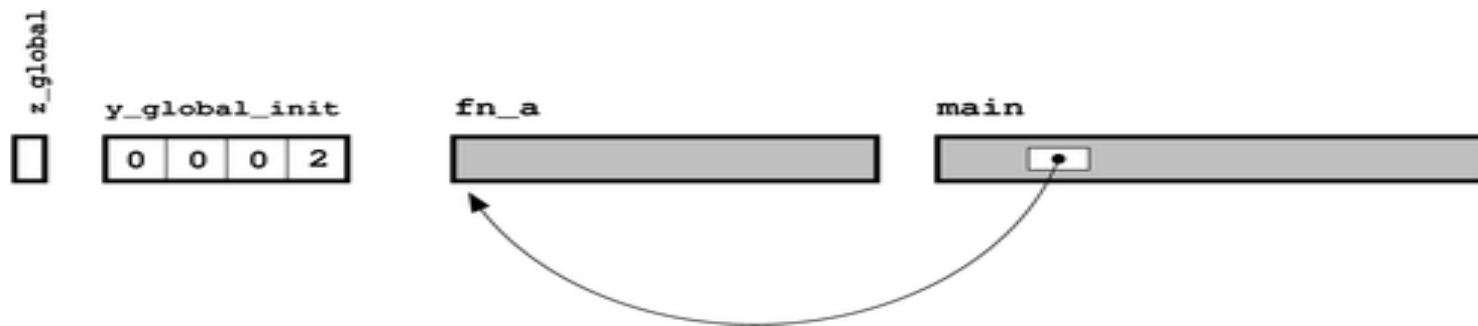
# Code Example (Last segment)

```
/* Initialized global variable */
int z_global = 11;
/* Second global named y_global_init, but they are both static */
static int y_global_init = 2;
/* Declaration of another global variable */
extern int x_global_init;

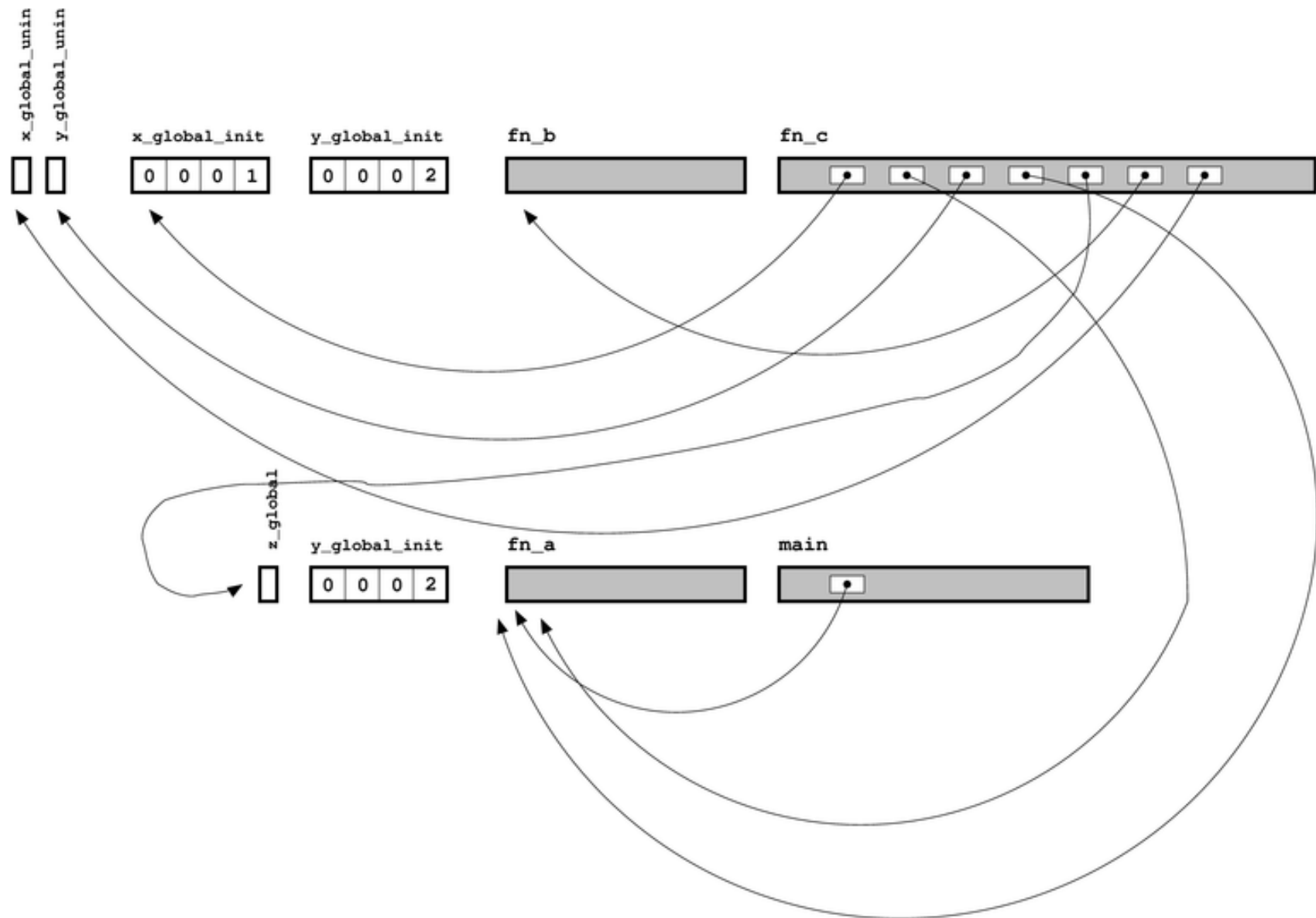
int fn_a(int x, int y)
{
    return(x+y);
}

int main(int argc, char *argv[])
{
    const char *message = "Hello, world";

    return fn_a(11,12);
}
```



# Linker



# Dissecting An Object File

- ❑ A class of **U** indicates an **undefined reference**, one of the "blanks" mentioned previously For this object, there are two: "fn\_a" and "z\_global" (Some versions of nm may also print out a *section*, which will be **\*UND\*** or **UNDEF** in this case)
- ❑ A class of **t** or **T** indicates where **code is defined**; the different classes indicate whether the **function is local to this file (t) or not (T)**—ie whether the function was originally declared with static Again, some systems may also show a section, something like **text**
- ❑ A class of **d** or **D** indicates an **initialized global variable**, and again the particular class indicates whether the **variable is local (d) or not (D)** If there's a section, it will be something like **data**
- ❑ For an **uninitialized global variable**, we get **b** if it's static/local, and **B or C** when it's not The section in this case will probably be something like **bss** or **\*COM\***



# Dissecting An Object File

- ❑ The key command is nm, which lists symbols in an object file in UNIX
  - ❑ On Windows, dumpbin command with /symbols option is roughly equivalent
- ❑ Let's see what nm gives on the object file produced from above C file

Symbols from c\_parts.o:

Name	Value	Class	Type	Size	Line	Section
fn_a		U	NOTYPE			*UND*
z_global		U	NOTYPE			*UND*
fn_b	00000000	t	FUNC	00000009		.text
x_global_init	00000000	D	OBJECT	00000004		.data
y_global_uninit	00000000	b	OBJECT	00000004		.bss
x_global_uninit	00000004	C	OBJECT	00000004		*COM*
y_global_init	00000004	d	OBJECT	00000004		.data
fn_c	00000009	T	FUNC	00000055		.text

	Code	Data				
		Global		Local		Dynamic
		Initialized	Uninitialized	Initialized	Uninitialized	
<b>Declaration</b>	int fn(int x);	extern int x;	extern int x;	N/A	N/A	N/A
<b>Definition</b>	int fn(int x) { ... }	int x = 1; (at file scope)	int x; (at file scope)	int x = 1; (at function scope)	int x; (at function scope)	(int* p = malloc(sizeof(int));)

# Static Libraries

- ❑ The most basic incarnation of a library is a *static library*
- ❑ Static libraries share code by reusing object files
  
- ❑ On UNIX systems :
  - ❑ The command to produce a static library is normally `ar`
  - ❑ The library file that it produces typically has a `.a` extension
  - ❑ Library files are normally also prefixed with `"lib"`
  - ❑ So `"-lfred"` will pick up `"libfred.a"`
  
- ❑ On Windows, static libraries have a `LIB` extension
  - ❑ Produced by the `LIB` tool

# Static Libraries

- ❑ Linker builds a list of the symbols it hasn't been able to resolve yet
- ❑ When all of the explicitly specified objects are done with, the linker has another place to look for the symbols that are left on this unresolved list in the **library**
- ❑ If the unresolved symbol is defined in one of the objects in the library, then that object is **added in**, exactly as if the user had given it on the command line in the first place
- ❑ If some particular symbol's definition is needed, the *whole object* that contains that symbol's definition is **included**
- ❑ The newly added object may resolve one undefined reference, but it may well come with a whole collection of **new undefined** references of its own for the linker to resolve
- ❑ The libraries are consulted **only when** the normal linking is done, and they are processed *in order*, **left to right** **If an object pulled in from a library late in the link line needs a symbol from a library earlier in the link line, the linker won't automatically find it**

# Static Linking Example

- ❑ Let's suppose we have the following object files, and a link line that pulls in `a.o`, `b.o`, `-lx` and `-ly`

File	a.o	b.o	libx.a			liby.a		
Object	a.o	b.o	x1.o	x2.o	x3.o	y1.o	y2.o	y3.o
Definitions	a1, a2, a3	b1, b2	x11, x12, x13	x21, x22, x23	x31, x32	y11, y12	y21, y22	y31, y32
Undefined references	b2, x12	a3, y22	x23, y12	y11		y21		x31

- ❑ Once the linker has processed `a.o` and `b.o`, it will have resolved the references to `b2` and `a3`, leaving **x12** and **y22** as still undefined
- ❑ At this point, the linker checks the first library `libx.a` for these symbols and finds that it can pull in `x1.o` to satisfy the `x12` reference
- ❑ However, doing so also adds `x23` and `y12` to the list of undefined references (so the list is now **y22**, **x23** and **y12**)
- ❑ The linker is still dealing with `libx.a`, so the `x23` reference is easily satisfied, by also pulling in `x2.o` from `libx.a`

# Static Linking Example

- ❑ However, this also adds `y11` to the list of undefined (which is now **`y22`, `y12` and `y11`**)
- ❑ None of these can be resolved further using `libx.a`, so the linker moves on to `liby.a`
- ❑ Here, the same sort of process applies and the linker will pull in both of `y1.o` and `y2.o`
- ❑ The first of these adds a reference to **`y21`**, but since `y2.o` is being pulled in anyway, **that reference is easily resolved**
- ❑ The net of this process is that **all undefined references have been resolved, and some but not all of the objects in the libraries have been included into the final executable**

# Static Linking Example

- ❑ Notice that the situation would have been a little different if (say) **b.o** also **had a reference to y32**
- ❑ If this had been the case, the linking of `libx.a` would have worked the same, but the processing of `liby.a` **would also have pulled in y3.o**
- ❑ **Pulling in this object would have added x31 to the list of unresolved symbols, and the link would have failed**
- ❑ By this stage the linker has already finished with `libx.a` and would not find the definition (in `x3.o`) for this symbol

# Shared Libraries

- ❑ For popular libraries like the C standard library (normally `libc`), having a static library has an obvious disadvantage
- ❑ Every executable program has a copy of the same code
  - ❑ This can take up a lot of unnecessary disk space, if every single executable file has a copy of `printf` and `fopen` and suchlike
- ❑ A slightly less obvious disadvantage is that once a program has been statically linked, the code in it is fixed forever
  - ❑ If someone finds and fixes a bug in `printf`, then every program has to be linked again in order to pick up the fixed code
- ❑ To get around these and other problems, *shared libraries* were introduced (normally indicated by a `so` extension, or `dll` on Windows machines and `dylib` on Mac OS X)
- ❑ For these kinds of libraries, the normal command line linker doesn't necessarily join up all of the dots
  - ❑ Instead, the regular linker takes a kind of "IOU" note, and defers the payment of that note until the moment when the program is actually run

# Dynamically Loaded Libraries

- ❑ Using shared libraries means that the final link is deferred until the moment when the program is run
  - ❑ On modern systems, it's possible to defer linking to runtime
- ❑ This is done with a pair of system calls, `dlopen` and `dlsym` (the rough Windows equivalents of these are called `LoadLibrary` and `GetProcAddress`)
- ❑ The first of these takes the name of a shared library and loads it into the address space of the running process
- ❑ Of course, this **extra library may itself have undefined symbols**
  - ❑ this call to `dlopen` may also trigger loading of other shared libraries
- ❑ The `dlopen` also allows the choice of whether to **resolve references at the instant that the library is loaded** (`RTLD_NOW`), or **one by one as each undefined reference is hit** (`RTLD_LAZY`)

[http://compsci.hunter.cuny.edu/~sweiss/resources/software\\_libraries.pdf](http://compsci.hunter.cuny.edu/~sweiss/resources/software_libraries.pdf)



# Dynamically Loaded Libraries

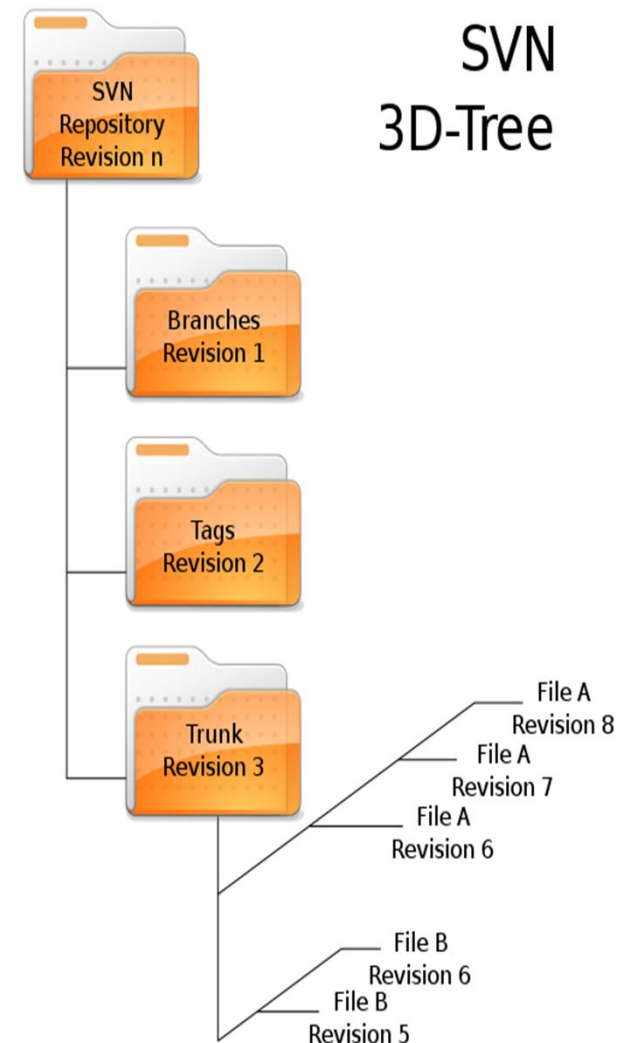
- ❑ The **first way** means that the **dlopen call takes much longer**, but the **second way involves the slight risk that the program later discovers an undefined reference that can't be resolved** (program i terminated)
- ❑ There's no way for a symbol from the dynamically loaded library to have a name
  - ❑ This is easily solved by adding an extra level of indirection, in this case, by using a pointer to the space for the symbol, rather than referring to it by name
  - ❑ The call `dlsym` takes a string parameter that gives the name of the symbol to be found, and returns a pointer to its location

# Versioning Systems

- Distributed
- Git
- Mercurial (Hg)
- Bazaar
- ...
- Client-Server
- Subversion (SVN)
- CVS
- ...
- Bitbucket (Mercurial or Git)
- Github (Git)
- Launchpad (Bazaar)
- Google Code (SVN, Mercurial or Git)
- Microsoft CodePlex (SVN, Mercurial or Git)
- Sourceforge (CVS, SVN, Bazaar, Git or Mercurial)
- Cornell Forge (SVN) - <http://forge.cornell.edu>

# Versioning & Revision Control - SVN

- ❑ **Apache Subversion** (often abbreviated **SVN**, after the command name `svn`) is a software versioning and revision control system
- ❑ Distributed as free sw under the Apache License
- ❑ **Maintain** current and historical versions of files such as source code, web pages, and docs
- ❑ Mostly compatible successor to the widely used Concurrent Versions System (**CVS**)
- ❑ Subversion is now a **top-level** Apache project used by a global community of contributors



# Conflict Problem

- ❑ Two solutions
  - ❑ The **locking** approach : preventing two developers to work on the same file
  - ❑ The **merge** approach : the second developer sending his changes to the server is responsible for merging his changes with the other developers changes already present on the server

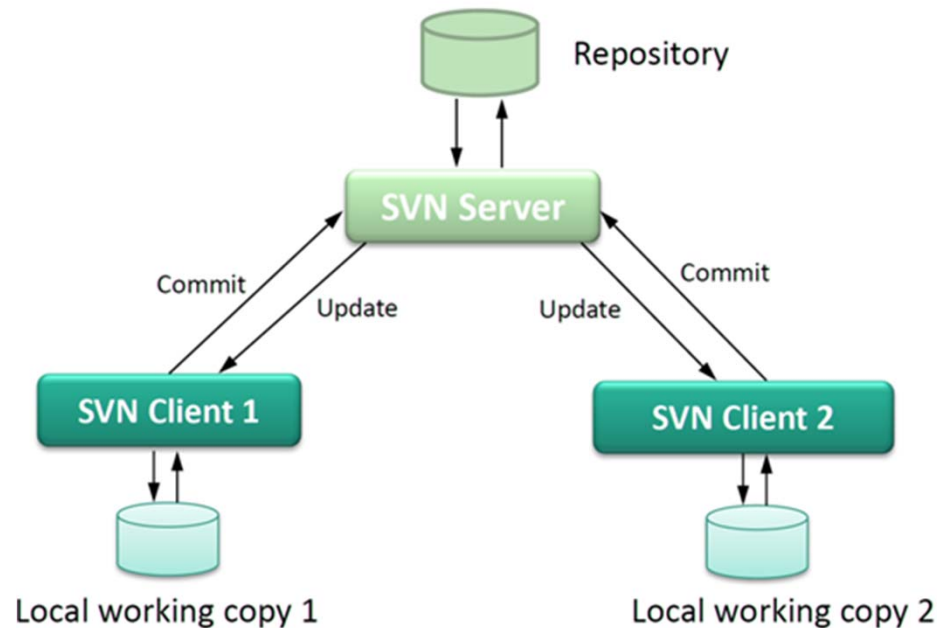
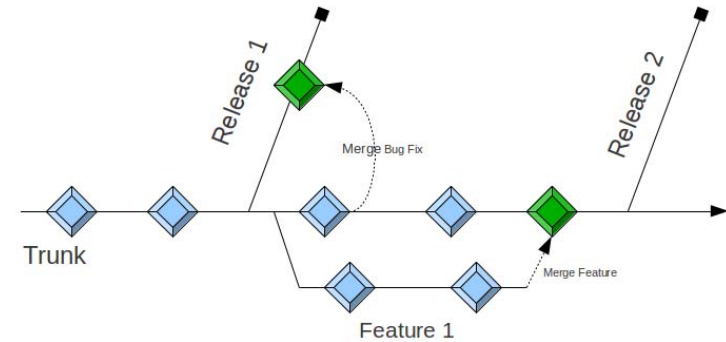
# SVN Goals

## ❑ Goals of a Version Control System

- ❑ Allow developers to work simultaneously
- ❑ Do not overwrite each other's changes
- ❑ Maintain history of every version of everything

## ❑ VCS is divided into two categories

- ❑ Centralized Version Control System (CVCS), means that it uses central server to store all files and enables team collaboration
- ❑ Distributed/Decentralized Version Control System (DVCS)



# SVN Terminology

## ❑ Terminology

- ❑ **Repository:** Heart of any version control system storing all work, including history. Repository is accessed over a network, acting as a server and version control tool acting as a client. Clients connect to the repository to store/retrieve changes to/from repository. By storing changes, a client makes these changes available to other people and by retrieving changes, a client takes other people's changes as a working copy
- ❑ **Trunk:** The trunk is a directory where all the main development happens and is usually checked out by developers to work on the project.
- ❑ **Branches:** Branch operation is used to create another line of development It is useful when you want your development process to fork off into two different directions. For example, when you release v50, you might want to create a branch so that development of v60 features can be kept separate from v50 bug-fixes
- ❑ **Tags :** The tags directory is used to store named snapshots of the project. Tag operation allows to give descriptive and memorable names to specific versions in the repository.

# SVN Terminology

## ❑ Terminology

- ❑ **Working copy:** Working copy is a snapshot of the repository. The repository is shared by all the teams, but people do not modify it directly. Instead, each developer checks out the working copy, a private workplace where developers do their work isolated from others.
- ❑ **Commit changes:** Commit is a process of storing changes from private workplace to central server. After commit, changes are made available to all the team. Other developers can retrieve these changes by updating their working copy. Commit is an atomic operation. Either the whole commit succeeds or is rolled back.

## ❑ Basic Commands

- ❑ `svn checkout svn+ssh://joe@lab.gr/repo/proj` //get current version
- ❑ `svn update` // bring changes from the repository into your working copy
- ❑ `svn lock file` // lock a file for update
- ❑ `svn unlock file` // unlock a file after update
- ❑ `svn diff -r 120` // display local modifications in a working copy vs revision 120
- ❑ `svn status` // print the status of working copy files and directories
- ❑ `svn log` //display history (comments, changes and details)
- ❑ `svn add "path"` // add a file or directory to your working copy
- ❑ `svn del "path"` // delete a file or directory from your working copy
- ❑ `svn commit -m "info"` // send changes from your working copy to the repository & tag info to new version number

# Git – Distributed VCS

- ❑ Recently, distributed version control instead of centralized
- ❑ Principles
  - ❑ No technically central repository, every copy is a repository
  - ❑ All developers can create local branches, share these branches with other developers without asking a central authority
  - ❑ Advanced branching and merging capabilities
  - ❑ More and more commonly used in free software projects (Linux kernel, X.org, etc.)
  - ❑ Most commonly used tools : Git, Mercurial



# Git

- ❑ `git config --global user.name "My Name" :`
- ❑ `git config --global user.email " myemail@some.domain"`
- ❑ `git init` // initialize working dir as repo
- ❑ `git clone http://mygiturl.edu` //clone a remote
- ❑ `git status` // show state of working dir
- ❑ `git add myfile1 myfile2` // add files
- ❑ `git add -A .` // add all modified files recursively
- ❑ `git commit -m "Some message"` // commit changes to local repo
- ❑ `git log` // view repo commit history
- ❑ `git diff` //show difference between current & last saved state
- ❑ `git diff HEAD 42 myfile` // compare to earlier version

# Git

- ❑ `git reset --hard HEAD` // erase all changes since last commit
- ❑ `git checkout 120 path/to/file` //restore file to prior revision
- ❑ `git remote add nickname remotespecurl` // add remote repo
- ❑ `git remote -v` // list associated remote git repos
- ❑ `git push nickname master` // commit local state to master branch of remote repo
- ❑ `git push -u nickname master` // push to remote repo if master does not exist
- ❑ `git pull nickname master` // get changes from remote repo

<http://git-scm.com/documentation>

# Diff & Patch (Local Copies)

- ❑ patch(1):
  - ❑ applies a diff(1) file (aka patch) to an original
  - ❑ may back up original file
  - ❑ may guess correct format
  - ❑ ignores leading or trailing “garbage”
  - ❑ allows for reversing the patch
  - ❑ may even correct context line numbers

# Example Diff & Patch

```
$ diff -ybw Makefile.2 Makefile.5
```

```
[...]
```

```
$ diff -u Makefile.2 Makefile.5 > /tmp/patch
```

```
$ patch < /tmp/patch
```

diff & merge using meld

# Tape Archive – TAR

- ❑ "tar" stands for *tape archive*, an archiving file format, originally developed in the early days of Unix
- ❑ The **tar** program is used to **create**, **maintain**, **modify**, and **extract** files that are archived in the **tar** format while preserving file system attributes such as:
  - ❑ user and group permissions
  - ❑ access and modification dates
  - ❑ directory structures

- ❑ Syntax

```
tar [-] A --catenate --concatenate | c --create | d --diff --compare |  
    --delete | r --append | t --list | --test-label | u --update |  
    x --extract --get [options] [pathname ...]
```

- ❑ Compressing

- ❑ `tar -zcvf file.tar.gz directory-name`

- ❑ Extracting

- ❑ `tar xvf file.tar`
  - ❑ `tar zxvf file.tar.gz`
  - ❑ `tar xjf file.tar.bz2`

# File (Un) Compression

Very useful for shrinking huge files and saving space

- ❑ `g[un]zip <file>`

- ❑ GNU zip compression utility. Creates .gz files.

- ❑ Ordinary performance (similar to zip)

- ❑ `b[un]zip2 <file>`

- ❑ More recent and effective compression utility.

- ❑ Creates .bz2 files. Usually 20-25% better than gzip

- ❑ `[un]lzma <file>`

- ❑ Much better compression ratio than bzip2 (up to 10 to 20%).

- ❑ Compatible command line options

# Local/Remote Backups

- ❑ Backup home directory, using rsync

- ❑ `rsync -azhe ssh --delete-during --info=progress /home/user/* user@ip_address:/home/user`

- ❑ Similar but slower using scp

- ❑ `scp -r /home/user user@ip_address:/home/user`

- ❑ Remote login via ssh to check...

# Related Commands

- ❑ `file, stat (lstat, fstat). size`
- ❑ `nm`
- ❑ `gdb`
- ❑ `objdump (for code, data)`
- ❑ `readelf, hexdump (for data)`
- ❑ `strip (symbols)`
- ❑ `binwalk, binvis.io`
- ❑ `buildroot`
- ❑ `ldconfig`
- ❑ `ldd`
- ❑ `...`
- ❑ `strace, ltrace, ftrace, valgrind, tracealyzer`
- ❑ `addr2line`



# The End

□ Questions??