2. Types of Computer Software

Contents

Chapter 1

Operating system

An **operating system** (**OS**) is software that manages computer hardware and software resources and provides common services for computer programs. The operating system is an essential component of the system software in a computer system. Application programs usually require an operating system to function.

Time-sharing operating systems schedule tasks for efficient use of the system and may also include accounting software for cost allocation of processor time, mass storage, printing, and other resources.

For hardware functions such as input and output and memory allocation, the operating system acts as an intermediary between programs and the computer hardware,[1][2] although the application code is usually executed directly by the hardware and frequently makes a system call to an OS function or be interrupted by it. Operating systems can be found on many devices that contain a computer—from cellular phones and video game consoles to supercomputers and web servers.

Examples of popular modern operating systems include Android, BSD, iOS, Linux, OS X, QNX, Microsoft Windows,[3] Windows Phone, and IBM z/OS. All these examples, except Windows, Windows Phone and z/OS, share roots in UNIX.

1.1 Types of operating systems

1.1.1 Single- and multi-tasking

A single-tasking system can only run one program at a time, while a multi-tasking operating system allows more than one program to be running in concurrency. This is achieved by time-sharing, dividing the available processor time between multiple processes which are each interrupted repeatedly in time-slices by a task scheduling subsystem of the operating system. Multi-tasking may be characterized in pre-emptive and co-operative types. In pre-emptive multitasking, the operating system slices the CPU time and dedicates a slot to each of the programs. Unix-like operating systems, e.g., Solaris, Linux, as well as AmigaOS support pre-emptive multitasking. Cooperative multitasking is achieved by relying on each process to provide time to the other processes in a defined manner. 16-bit versions of Microsoft Windows used cooperative multi-tasking. 32-bit versions of both Windows NT and Win9x, used pre-emptive multi-tasking. Mac OS prior to OS X used to support cooperative multitasking.

1.1.2 Single- and multi-user

Single-user operating systems have no facilities to distinguish users, but may allow multiple programs to run at the same time. A multi-user operating system extends the basic concept of multi-tasking with facilities that identify processes and resources, such as disk space, belonging to multiple users, and the system permits multiple users to interact with the system at the same time. Time-sharing operating systems schedule tasks for efficient use of the system and may also include accounting software for cost allocation of processor time, mass storage, printing, and other resources to multiple users.

1.1.3 Distributed

A distributed operating system manages a group of distinct computers and makes them appear to be a single computer. The development of networked computers that could be linked and communicate with each other gave rise to distributed computing. Distributed computations are carried out on more than one machine. When computers in a group work in cooperation, they form a distributed system.

1.1.4 Templated

In an OS, distributed and cloud computing context, templating refers to creating a single virtual machine image as a guest operating system, then saving it as a tool for multiple running virtual machines (Gagne, 2012, p. 716). The technique is used both in virtualization and cloud computing management, and is common in large server warehouses. [4]

1.1.5 Embedded

Embedded operating systems are designed to be used in embedded computer systems. They are designed to operate on small machines like PDAs with less autonomy. They are able to operate with a limited number of resources. They are very compact and extremely efficient by design. Windows CE and Minix 3 are some examples of embedded operating systems.

1.1.6 Real-time

A real-time operating system is an operating system that guaranties to process events or data within a certain short amount of time. A real-time operating system may be single- or multi-tasking, but when multitasking, it uses specialized scheduling algorithms so that a deterministic nature of behavior is achieved. An event-driven system switches between tasks based on their priorities or external events while time-sharing operating systems switch tasks based on clock interrupts.

1.2 History

Main article: History of operating systems
See also: Resident monitor

Early computers were built to perform a series of single tasks, like a calculator. Basic operating system features were developed in the 1950s, such as resident monitor functions that could automatically run different programs in succession to speed up processing. Operating systems did not exist in their modern and more complex forms until the early 1960s.[5] Hardware features were added, that enabled use of runtime libraries, interrupts, and parallel processing. When personal computers became popular in the 1980s, operating systems were made for them similar in concept to those used on larger computers.

In the 1940s, the earliest electronic digital systems had no operating systems. Electronic systems of this time were programmed on rows of mechanical switches or by jumper wires on plug boards. These were special-purpose systems that, for example, generated ballistics tables for the military or controlled the printing of payroll checks from data on punched paper cards. After programmable general purpose computers were invented, machine languages (consisting of strings of the binary digits 0 and 1 on punched paper tape) were introduced that sped up the programming process (Stern, 1981).

In the early 1950s, a computer could execute only one program at a time. Each user had sole use of the computer for a limited period of time and would arrive at a scheduled time with program and data on punched paper cards and/or punched tape. The program would be loaded into the machine, and the machine would be set to

OS/360 was used on most IBM mainframe computers beginning in 1966, including computers utilized by the Apollo program.

work until the program completed or crashed. Programs could generally be debugged via a front panel using toggle switches and panel lights. It is said that Alan Turing was a master of this on the early Manchester Mark 1 machine, and he was already deriving the primitive conception of an operating system from the principles of the Universal Turing machine.[5]

Later machines came with libraries of programs, which would be linked to a user's program to assist in operations such as input and output and generating computer code from human-readable symbolic code. This was the genesis of the modern-day operating system. However, machines still ran a single job at a time. At Cambridge University in England the job queue was at one time a washing line from which tapes were hung with different colored clothes-pegs to indicate job-priority.

An improvement was the Atlas Supervisor introduced with the Manchester Atlas commissioned in 1962, 'considered by many to be the first recognisable modern operating system'.[6] Brinch Hansen described it as "the most significant breakthrough in the history of operating systems."[7]

1.2.1 Mainframes

Main article: Mainframe computer
See also: History of IBM mainframe operating systems

Through the 1950s, many major features were pioneered in the field of operating systems, including batch processing, input/output interrupt, buffering, multitasking, spooling, runtime libraries, link-loading, and programs for sorting records in files. These features were included or not included in application software at the option of application programmers, rather than in a separate operating system used by all applications. In 1959, the SHARE Operating System was released as an integrated utility for the IBM 704, and later in the 709 and 7090 mainframes, although it was quickly supplanted by IBSYS/IBJOB on the 709, 7090 and 7094.

During the 1960s, IBM's OS/360 introduced the concept of a single OS spanning an entire product line, which was crucial for the success of the System/360 machines. IBM's current mainframe operating systems are distant descendants of this original system and applications written for OS/360 can still be run on modern machines.

OS/360 also pioneered the concept that the operating system keeps track of all of the system resources that are used, including program and data space allocation in main memory and file space in secondary storage, and file locking during update. When the process is terminated for any reason, all of these resources are re-claimed by the operating system.

The alternative CP-67 system for the S/360-67 started a whole line of IBM operating systems focused on the concept of virtual machines. Other operating systems used on IBM S/360 series mainframes included systems developed by IBM: COS/360 (Compatibility Operating System), DOS/360 (Disk Operating System), TSS/360 (Time Sharing System), TOS/360 (Tape Operating System), BOS/360 (Basic Operating System), and ACP (Airline Control Program), as well as a few non-IBM systems: MTS (Michigan Terminal System), MUSIC (Multi-User System for Interactive Computing), and ORVYL (Stanford Timesharing System).

Control Data Corporation developed the SCOPE operating system in the 1960s, for batch processing. In cooperation with the University of Minnesota, the Kronos and later the NOS operating systems were developed during the 1970s, which supported simultaneous batch and timesharing use. Like many commercial timesharing systems, its interface was an extension of the Dartmouth BASIC operating systems, one of the pioneering efforts in timesharing and programming languages. In the late 1970s, Control Data and the University of Illinois developed the PLATO operating system, which used plasma panel displays and long-distance time sharing networks. Plato was remarkably innovative for its time, featuring real-time chat, and multi-user graphical games.

In 1961, Burroughs Corporation introduced the B5000 with the MCP, (Master Control Program) operating system. The B5000 was a stack machine designed to exclusively support high-level languages with no machine language or assembler, and indeed the MCP was the first OS to be written exclusively in a high-level language – ESPOL, a dialect of ALGOL. MCP also introduced many other ground-breaking innovations, such as being the first commercial implementation of virtual memory. During development of the AS400, IBM made an approach to Burroughs to licence MCP to run on the AS400 hardware. This proposal was declined by Burroughs management to protect its existing hardware production. MCP is still in use today in the Unisys ClearPath/MCP line of computers.

UNIVAC, the first commercial computer manufacturer, produced a series of EXEC operating systems. Like all early main-frame systems, this batch-oriented system managed magnetic drums, disks, card readers and line printers. In the 1970s, UNIVAC produced the Real-Time Basic (RTB) system to support large-scale time sharing, also patterned after the Dartmouth BC system.

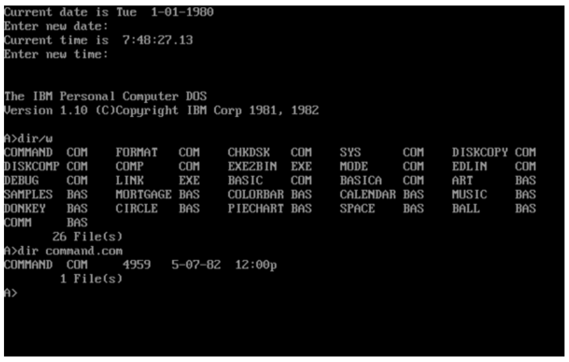
General Electric and MIT developed General Electric Comprehensive Operating Supervisor (GECOS), which introduced the concept of ringed security privilege levels. After acquisition by Honeywell it was renamed General Comprehensive Operating System (GCOS).

Digital Equipment Corporation developed many operating systems for its various computer lines, including TOPS-10 and TOPS-20 time sharing systems for the 36-bit PDP-10 class systems. Before the widespread use of UNIX, TOPS-10 was a particularly popular system in universities, and in the early ARPANET community.

From the late 1960s through the late 1970s, several hardware capabilities evolved that allowed similar or ported software to run on more than one system. Early systems had utilized microprogramming to implement features on their systems in order to permit different underlying computer architectures to appear to be the same as others in a series. In fact, most 360s after the 360/40 (except the 360/165 and 360/168) were microprogrammed implementations.

The enormous investment in software for these systems made since the 1960s caused most of the original computer manufacturers to continue to develop compatible operating systems along with the hardware. Notable supported mainframe operating systems include:

- Burroughs MCP – B5000, 1961 to Unisys Clearpath/MCP, present.
- IBM OS/360 – IBM System/360, 1966 to IBM z/OS, present.
- IBM CP-67 – IBM System/360, 1967 to IBM z/VM, present.
- UNIVAC EXEC 8 – UNIVAC 1108, 1967, to OS 2200 Unisys Clearpath Dorado, present.

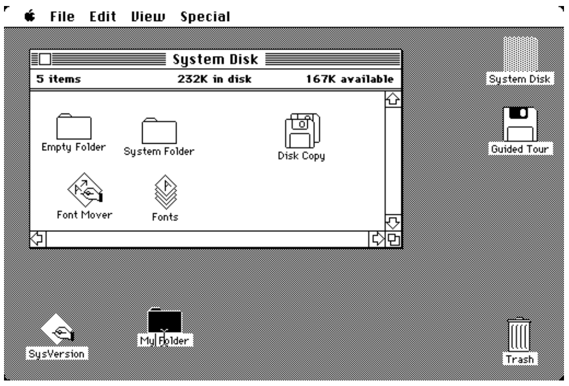


```
Current date is Tue  1-01-1980
Enter new date:
Current time is  7:48:27.13
Enter new time:


The IBM Personal Computer DOS
Version 1.10 (C)Copyright IBM Corp 1981, 1982

A>dir/w
COMMAND  COM    FORMAT   COM    CHKDSK   COM    SYS      COM    DISKCOPY COM
DISKCOMP COM    COMP     COM    EXE2BIN  EXE    MODE     COM    EDLIN    COM
DEBUG    COM    LINK     EXE    BASIC    COM    BASICA   COM    ART      BAS
SAMPLES  BAS    MORTGAGE BAS    COLORBAR BAS    CALENDAR BAS    MUSIC    BAS
DONKEY   BAS    CIRCLE   BAS    PIECHART BAS    SPACE    BAS    BALL     BAS
COMM     BAS
       26 File(s)
A>dir command.com
COMMAND  COM     4959   5-07-82  12:00p
        1 File(s)
A>
```

PC DOS was an early personal computer OS that featured a command line interface.



Mac OS by Apple Computer became the first widespread OS to feature a graphical user interface. Many of its features such as windows and icons would later become commonplace in GUIs.

1.2.2 Microcomputers

The first microcomputers did not have the capacity or need for the elaborate operating systems that had been developed for mainframes and minis; minimalistic operating systems were developed, often loaded from ROM and known as *monitors*. One notable early disk operating system was CP/M, which was supported on many early microcomputers and was closely imitated by Microsoft's MS-DOS, which became widely popular as the operating system chosen for the IBM PC (IBM’s version of it was called IBM DOS or PC DOS). In the '80s, Apple Computer Inc. (now Apple Inc.) abandoned its popular Apple II series of microcomputers to introduce the Apple Macintosh computer with an innovative Graphical User Interface (GUI) to the Mac OS.

The introduction of the Intel 80386 CPU chip with 32-bit architecture and paging capabilities, provided personal computers with the ability to run multitasking operating systems like those of earlier minicomputers and mainframes. Microsoft responded to this progress by hiring Dave Cutler, who had developed the VMS operating system for Digital Equipment Corporation. He would lead the development of the Windows NT operating system, which continues to serve as the basis for Microsoft’s operating systems line. Steve Jobs, a co-founder of Apple Inc., started NeXT Computer Inc., which developed the NEXTSTEP operating system. NEXTSTEP would later be acquired by Apple Inc. and used, along with code from FreeBSD as the core of Mac OS X.

The GNU Project was started by activist and programmer Richard Stallman with the goal of creating a complete free software replacement to the proprietary UNIX operating system. While the project was highly successful in duplicating the functionality of various parts of UNIX, development of the GNU Hurd kernel proved to be unproductive. In 1991, Finnish computer science student Linus Torvalds, with cooperation from volunteers collaborating over the Internet, released the first version of the Linux kernel. It was soon merged with the GNU user space components and system software to form a complete operating system. Since then, the combination of the two major components has usually been referred to as simply “Linux” by the software industry, a naming convention that Stallman and the Free Software Foundation remain opposed to, preferring the name GNU/Linux. The Berkeley Software Distribution, known as BSD, is the UNIX derivative distributed by the University of California, Berkeley, starting in the 1970s. Freely distributed and ported to many minicomputers, it eventually also gained a following for use on PCs, mainly as FreeBSD, NetBSD and OpenBSD.

1.3 Examples of operating systems

1.3.1 Unix and Unix-like operating systems

Evolution of Unix systems
Main article: Unix

Unix was originally written in assembly language.[8] Ken Thompson wrote B, mainly based on BCPL, based on his experience in the MULTICS project. B was replaced by C, and Unix, rewritten in C, developed into a large, complex family of inter-related operating systems which have been influential in every modern operating system (see History).

The *Unix-like* family is a diverse group of operating systems, with several major sub-categories including System V, BSD, and Linux. The name "UNIX" is a trademark of The Open Group which licenses it for use with any operating system that has been shown to conform to their definitions. “UNIX-like” is commonly used to refer to the large set of operating systems which resemble the original UNIX.

Unix-like systems run on a wide variety of computer architectures. They are used heavily for servers in business, as well as workstations in academic and engineering environments. Free UNIX variants, such as Linux and BSD, are popular in these areas.

Four operating systems are certified by The Open Group (holder of the Unix trademark) as Unix. HP's HP-UX and IBM's AIX are both descendants of the original System V Unix and are designed to run only on their respective vendor's hardware. In contrast, Sun Microsystems's Solaris Operating System can run on multiple types of hardware, including x86 and Sparc servers, and PCs. Apple's OS X, a replacement for Apple's earlier (non-Unix) Mac OS, is a hybrid kernel-based BSD variant derived from NeXTSTEP, Mach, and FreeBSD.

Unix interoperability was sought by establishing the POSIX standard. The POSIX standard can be applied to any operating system, although it was originally created for various Unix variants.

BSD and its descendants

Main article: Berkeley Software Distribution

A subgroup of the Unix family is the Berkeley Software Distribution family, which includes FreeBSD, NetBSD, and OpenBSD. These operating systems are most commonly found on webservers, although they can also function as a personal computer OS. The Internet owes much of its existence to BSD, as many of the protocols now commonly used by computers to connect, send and receive data over a network were widely implemented and refined in BSD. The World Wide Web was also first demonstrated on a number of computers running an OS based on BSD called NextStep.

The first server for the World Wide Web ran on NeXTSTEP, based on BSD

BSD has its roots in Unix. In 1974, University of California, Berkeley installed its first Unix system. Over time, students and staff in the computer science department there began adding new programs to make things easier, such as text editors. When Berkeley received new VAX computers in 1978 with Unix installed, the school's undergraduates modified Unix even more in order to take advantage of the computer's hardware possibilities. The Defense Advanced Research Projects Agency of the US Department of Defense took interest, and decided to fund the project. Many schools, corporations, and government organizations took notice and started to use Berkeley's version of Unix instead of the official one distributed by AT&T.

Steve Jobs, upon leaving Apple Inc. in 1985, formed NeXT Inc., a company that manufactured high-end computers running on a variation of BSD called NeXTSTEP. One of these computers was used by Tim Berners-Lee as the first webserver to create the World Wide Web.

Developers like Keith Bostic encouraged the project to replace any non-free code that originated with Bell Labs. Once this was done, however, AT&T sued. Eventually, after two years of legal disputes, the BSD project came out ahead and spawned a number of free derivatives, such as FreeBSD and NetBSD.

OS X

Main article: OS X

OS X (formerly "Mac OS X") is a line of open core graphical operating systems developed, marketed, and sold by Apple Inc., the latest of which is pre-loaded on all currently shipping Macintosh computers. OS X is the successor to the original Mac OS, which had been Apple's primary operating system since 1984. Unlike its predecessor, OS X is a UNIX operating system built on technology that had been developed at NeXT through the second half of the 1980s and up until Apple purchased the company in early 1997. The operating system was first released in 1999 as Mac OS X Server 1.0, with a desktop-oriented version (Mac OS X v10.0 "Cheetah") following in March 2001. Since then, six more distinct "client" and "server" editions of OS X have been released, until the two were merged in OS X 10.7 "Lion". Releases of OS X v10.0 through v10.8 are named after big cats. Starting with v10.9, "Mavericks", OS X versions are named after inspirational places in California.[9] OS X 10.10 "Yosemite", the most recent version, was announced and released on June 2, 2014 at the WWDC 2014.

The standard user interface of OS X

Prior to its merging with OS X, the server edition – OS X Server – was architecturally identical to its desktop counterpart and usually ran on Apple's line of Macintosh server hardware. OS X Server included work group management and administration software tools that provide simplified access to key network services, including a mail transfer agent, a Samba server, an LDAP server, a domain name server, and others. With Mac OS X v10.7 Lion, all server aspects of Mac OS X Server have been integrated into the client version and the product re-branded as “OS X” (dropping “Mac” from the name). The server tools are now offered as an application.[10]

Linux and GNU

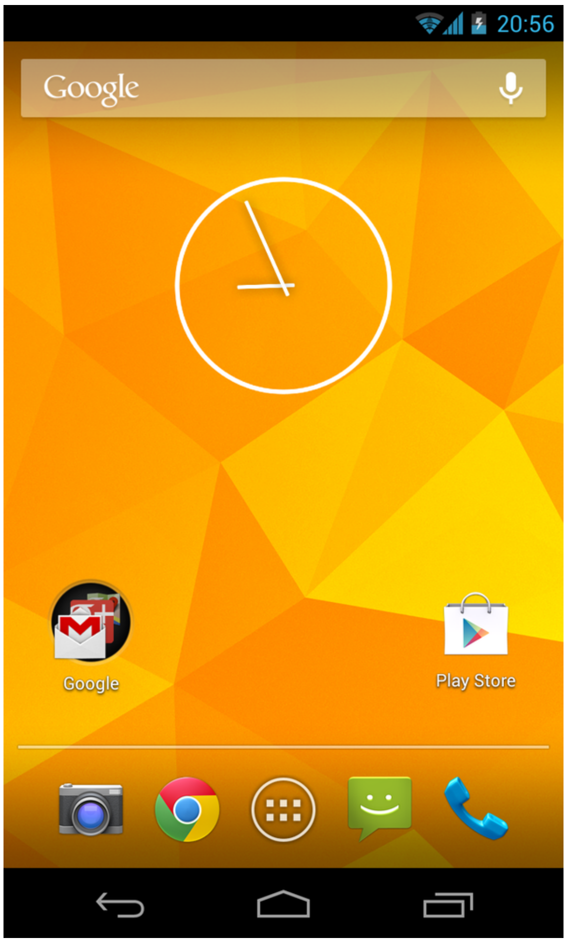
Main articles: GNU, Linux and Linux kernel

The GNU project is a collaboration of many program-

Ubuntu, desktop GNU/Linux distribution

mers who envisioned to create a free and open operating system that was similar to Unix but with new code licensed on the open-source license model. It was started in 1983 by Richard Stallman, and is responsible for many components of most Linux variants. Thousands of pieces of software for virtually every operating system are licensed under the GNU General Public License. Meanwhile, the Linux kernel originated in 1991 as a side project of Linus Torvalds, while a university student in Finland. He posted information about his project on a newsgroup for computer students and programmers, and received support and assistance from volunteers who succeeded in creating a complete and functional kernel. GNU programmers joint the effort and both groups worked to integrate the finished GNU parts with the Linux kernel to create a complete operating system.

Linux is Unix-like, but was developed without any Unix code, unlike BSD and its variants. Because of its open license model, the Linux kernel code is available for study and modification, which resulted in its use on a wide range of computing machinery from supercomputers to smart-watches. Although estimates suggest that Linux and GNU software are used on only 1.82% of all personal computers,[11][12] it has been widely adopted for use in servers[13] and embedded systems[14] such as cell phones. GNU/Linux has superseded Unix on many platforms and is used on the ten most powerful supercomputers in the world.[15] The Linux kernel is used in some popular distributions, such as Red Hat, Debian, Ubuntu, Linux Mint and Google's Android.



Android, a popular mobile operating system using the Linux kernel

Google Chromium OS Main article: Google Chromium OS

Chromium is an operating system based on the Linux kernel and designed by Google. Since Chromium OS targets computer users who spend most of their time on the Internet, it is mainly a web browser with limited ability to run local applications, though it has a built-in file manager and media player. Instead, it relies on Internet applications (or Web apps) used in the web browser to accomplish tasks such as word processing.[16] Chromium OS differs from Chrome OS in that Chromium is open-source and used primarily by developers whereas Chrome OS is the operating system shipped out in Chromebooks.[17]

1.3.2 Microsoft Windows

Main article: Microsoft Windows

Microsoft Windows is a family of proprietary operating systems designed by Microsoft Corporation and primarily targeted to Intel architecture based computers, with an estimated 88.9 percent total usage share on Web connected computers.[12][18][19][20] The newest version is Windows 8.1 for workstations and Windows Server 2012 R2 for servers. Windows 7 recently overtook Windows XP as most used OS.[21][22][23]

Microsoft Windows originated in 1985 as an operating environment running on top of MS-DOS, which was the standard operating system shipped on most Intel architecture personal computers at the time. In 1995, Windows 95 was released which only used MS-DOS as a bootstrap. For backwards compatibility, Win9x could run real-mode MS-DOS[24][25] and 16 bits Windows 3.x[26] drivers. Windows ME, released in 2000, was the last version in the Win9x family. Later versions have all been based on the Windows NT kernel. Current client versions of Windows run on IA-32, x86-64 and 32-bit ARM microprocessors.[27] In addition Itanium is still supported in older server version Windows Server 2008 R2. In the past, Windows NT supported additional architectures.

Server editions of Windows are widely used. In recent years, Microsoft has expended significant capital in an effort to promote the use of Windows as a server operating system. However, Windows' usage on servers is not as widespread as on personal computers, as Windows competes against Linux and BSD for server market share.[28][29]The first PC that used windows operating system was the IBM Personal System/2.

1.3.3 Other

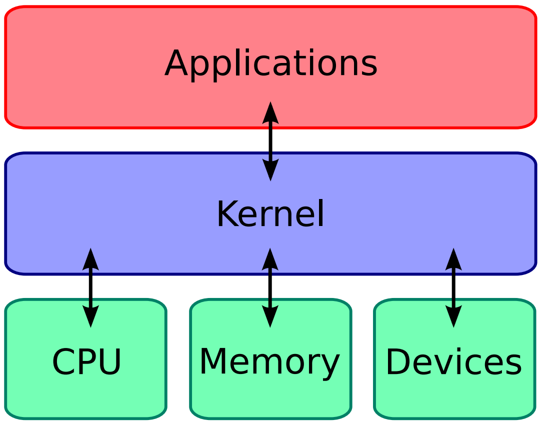
There have been many operating systems that were significant in their day but are no longer so, such as AmigaOS; OS/2 from IBM and Microsoft; Mac OS, the non-Unix precursor to Apple's Mac OS X; BeOS; XTS-300; RISC OS; MorphOS; Haiku; BareMetal and FreeMint. Some are still used in niche markets and continue to be developed as minority platforms for enthusiast communities and specialist applications. OpenVMS formerly from DEC, is still under active development by Hewlett-Packard. Yet other operating systems are used almost exclusively in academia, for operating systems education or to do research on operating system concepts. A typical example of a system that fulfills both roles is MINIX, while for example Singularity is used purely for research.

Other operating systems have failed to win significant market share, but have introduced innovations that have influenced mainstream operating systems, not least Bell Labs' Plan 9.

1.4 Components

The components of an operating system all exist in order to make the different parts of a computer work together. All user software needs to go through the operating system in order to use any of the hardware, whether it be as simple as a mouse or keyboard or as complex as an Internet component.

1.4.1 Kernel



A kernel connects the application software to the hardware of a computer.

Main article: Kernel (computing)

With the aid of the firmware and device drivers, the kernel provides the most basic level of control over all of the computer's hardware devices. It manages memory access for programs in the RAM, it determines which programs get access to which hardware resources, it sets up or resets the CPU's operating states for optimal operation at all times, and it organizes the data for long-term non-volatile storage with file systems on such media as disks, tapes, flash memory, etc.

Program execution

Main article: Process (computing)

The operating system provides an interface between an application program and the computer hardware, so that an application program can interact with the hardware only by obeying rules and procedures programmed into the operating system. The operating system is also a set of services which simplify development and execution of application programs. Executing an application program involves the creation of a process by the operating system kernel which assigns memory space and other resources, establishes a priority for the process in multi-tasking systems, loads program binary code into memory, and initiates execution of the application program which then interacts with the user and with hardware devices.

Interrupts

Main article: Interrupt

Interrupts are central to operating systems, as they provide an efficient way for the operating system to interact with and react to its environment. The alternative — having the operating system "watch" the various sources of input for events (polling) that require action — can be found in older systems with very small stacks (50 or 60 bytes) but is unusual in modern systems with large stacks. Interrupt-based programming is directly supported by most modern CPUs. Interrupts provide a computer with a way of automatically saving local register contexts, and running specific code in response to events. Even very basic computers support hardware interrupts, and allow the programmer to specify code which may be run when that event takes place.

When an interrupt is received, the computer's hardware automatically suspends whatever program is currently running, saves its status, and runs computer code previously associated with the interrupt; this is analogous to placing a bookmark in a book in response to a phone call. In modern operating systems, interrupts are handled by the operating system's kernel. Interrupts may come from either the computer's hardware or the running program.

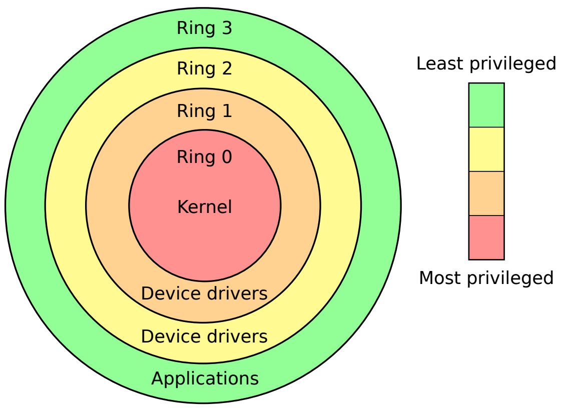
When a hardware device triggers an interrupt, the operating system's kernel decides how to deal with this event, generally by running some processing code. The amount of code being run depends on the priority of the interrupt (for example: a person usually responds to a smoke detector alarm before answering the phone). The processing of hardware interrupts is a task that is usually delegated to software called a device driver, which may be part of the operating system's kernel, part of another program, or both. Device drivers may then relay information to a running program by various means.

A program may also trigger an interrupt to the operating system. If a program wishes to access hardware, for example, it may interrupt the operating system's kernel, which causes control to be passed back to the kernel. The kernel then processes the request. If a program wishes additional resources (or wishes to shed resources) such as memory, it triggers an interrupt to get the kernel's attention.

Modes

Main articles: Protected mode and Supervisor mode

Modern CPUs support multiple modes of operation. CPUs with this capability use at least two modes: protected mode and supervisor mode. The supervisor mode is used by the operating system's kernel for low level tasks that need unrestricted access to hardware, such as controlling how memory is written and erased, and communication with devices like graphics cards. Protected mode, in contrast, is used for almost everything else. Applications operate within protected mode, and can only use hardware by communicating with the kernel, which controls everything in supervisor mode. CPUs might have other modes similar to protected mode as well, such as the virtual modes in order to emulate older processor types, such as 16-bit processors on a 32-bit one, or 32-bit processors on a 64-bit one.



Privilege rings for the x86 available in protected mode. Operating systems determine which processes run in each mode.

When a computer first starts up, it is automatically running in supervisor mode. The first few programs to run on the computer, being the BIOS or EFI, bootloader, and the operating system have unlimited access to hardware – and this is required because, by definition, initializing a protected environment can only be done outside of one. However, when the operating system passes control to another program, it can place the CPU into protected mode.

In protected mode, programs may have access to a more limited set of the CPU's instructions. A user program may leave protected mode only by triggering an interrupt, causing control to be passed back to the kernel. In this way the operating system can maintain exclusive control over things like access to hardware and memory.

The term "protected mode resource" generally refers to one or more CPU registers, which contain information that the running program isn't allowed to alter. Attempts to alter these resources generally causes a switch to supervisor mode, where the operating system can deal with the illegal operation the program was attempting (for example, by killing the program).

Memory management

Main article: Memory management

Among other things, a multiprogramming operating system kernel must be responsible for managing all system memory which is currently in use by programs. This ensures that a program does not interfere with memory already in use by another program. Since programs time share, each program must have independent access to memory.

Cooperative memory management, used by many early operating systems, assumes that all programs make voluntary use of the kernel's memory manager, and do not exceed their allocated memory. This system of memory management is almost never seen any more, since programs often contain bugs which can cause them to exceed their allocated memory. If a program fails, it may cause memory used by one or more other programs to be affected or overwritten. Malicious programs or viruses may purposefully alter another program's memory, or may affect the operation of the operating system itself. With cooperative memory management, it takes only one misbehaved program to crash the system.

Memory protection enables the kernel to limit a process' access to the computer's memory. Various methods of memory protection exist, including memory segmentation and paging. All methods require some level of hardware support (such as the 80286 MMU), which doesn't exist in all computers.

In both segmentation and paging, certain protected mode registers specify to the CPU what memory address it should allow a running program to access. Attempts to access other addresses trigger an interrupt which cause the CPU to re-enter supervisor mode, placing the kernel in charge. This is called a segmentation violation or Seg-V for short, and since it is both difficult to assign a meaningful result to such an operation, and because it is usually a sign of a misbehaving program, the kernel generally resorts to terminating the offending program, and reports the error.

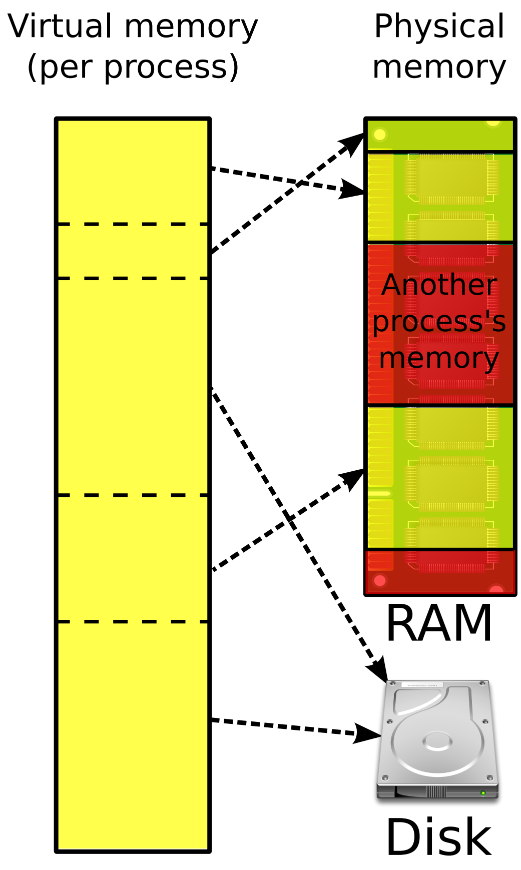
Windows versions 3.1 through ME had some level of memory protection, but programs could easily circumvent the need to use it. A general protection fault would be produced, indicating a segmentation violation had occurred; however, the system would often crash anyway.

Virtual memory

Main article: Virtual memory
Further information: Page fault
The use of virtual memory addressing (such as paging or segmentation) means that the kernel can choose what memory each program may use at any given time, allowing the operating system to use the same memory locations for multiple tasks.

If a program tries to access memory that isn't in its current range of accessible memory, but nonetheless has been allocated to it, the kernel is interrupted in the same way as it would if the program were to exceed its allocated memory. (See section on memory management.) Under UNIX this kind of interrupt is referred to as a page fault.

When the kernel detects a page fault it generally adjusts the virtual memory range of the program which triggered



Many operating systems can "trick" programs into using memory scattered around the hard disk and RAM as if it is one continuous chunk of memory, called virtual memory.

it, granting it access to the memory requested. This gives the kernel discretionary power over where a particular application's memory is stored, or even whether or not it has actually been allocated yet.

In modern operating systems, memory which is accessed less frequently can be temporarily stored on disk or other media to make that space available for use by other programs. This is called swapping, as an area of memory can be used by multiple programs, and what that memory area contains can be swapped or exchanged on demand.

"Virtual memory" provides the programmer or the user with the perception that there is a much larger amount of RAM in the computer than is really there.[30]

Multitasking

Main articles: Computer multitasking and Process management (computing)
Further information: Context switch, Preemptive multitasking and Cooperative multitasking

Multitasking refers to the running of multiple independent computer programs on the same computer; giving the appearance that it is performing the tasks at the same time. Since most computers can do at most one or two things at one time, this is generally done via time-sharing, which means that each program uses a share of the computer's time to execute.

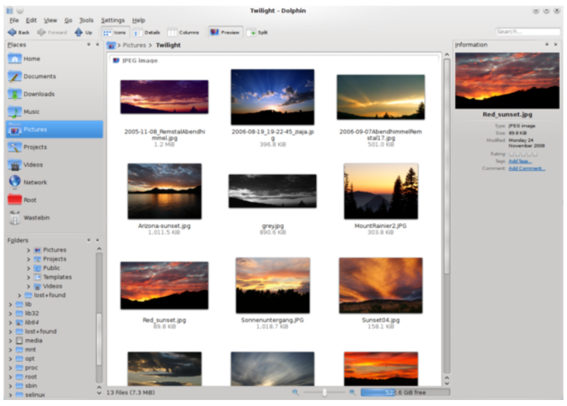
An operating system kernel contains a scheduling program which determines how much time each process spends executing, and in which order execution control should be passed to programs. Control is passed to a process by the kernel, which allows the program access to the CPU and memory. Later, control is returned to the kernel through some mechanism, so that another program may be allowed to use the CPU. This so-called passing of control between the kernel and applications is called a context switch.

An early model which governed the allocation of time to programs was called cooperative multitasking. In this model, when control is passed to a program by the kernel, it may execute for as long as it wants before explicitly returning control to the kernel. This means that a malicious or malfunctioning program may not only prevent any other programs from using the CPU, but it can hang the entire system if it enters an infinite loop.

Modern operating systems extend the concepts of application preemption to device drivers and kernel code, so that the operating system has preemptive control over internal run-times as well.

The philosophy governing preemptive multitasking is that of ensuring that all programs are given regular time on the CPU. This implies that all programs must be limited in how much time they are allowed to spend on the CPU without being interrupted. To accomplish this, modern operating system kernels make use of a timed interrupt. A protected mode timer is set by the kernel which triggers a return to supervisor mode after the specified time has elapsed. (See above sections on Interrupts and Dual Mode Operation.)

On many single user operating systems cooperative multitasking is perfectly adequate, as home computers generally run a small number of well tested programs. The AmigaOS is an exception, having pre-emptive multitasking from its very first version. Windows NT was the first version of Microsoft Windows which enforced preemptive multitasking, but it didn't reach the home user market until Windows XP (since Windows NT was targeted at professionals).



Filesystems allow users and programs to organize and sort files on a computer, often through the use of directories (or "folders")

Disk access and file systems

Main article: Virtual file system

Access to data stored on disks is a central feature of all operating systems. Computers store data on disks using files, which are structured in specific ways in order to allow for faster access, higher reliability, and to make better use out of the drive's available space. The specific way in which files are stored on a disk is called a file system, and enables files to have names and attributes. It also allows them to be stored in a hierarchy of directories or folders arranged in a directory tree.

Early operating systems generally supported a single type of disk drive and only one kind of file system. Early file systems were limited in their capacity, speed, and in the kinds of file names and directory structures they could use. These limitations often reflected limitations in the operating systems they were designed for, making it very difficult for an operating system to support more than one file system.

While many simpler operating systems support a limited range of options for accessing storage systems, operating systems like UNIX and Linux support a technology known as a virtual file system or VFS. An operating system such as UNIX supports a wide array of storage devices, regardless of their design or file systems, allowing them to be accessed through a common application programming interface (API). This makes it unnecessary for programs to have any knowledge about the device they are accessing. A VFS allows the operating system to provide programs with access to an unlimited number of devices with an infinite variety of file systems installed on them, through the use of specific device drivers and file system drivers.

A connected storage device, such as a hard drive, is accessed through a device driver. The device driver understands the specific language of the drive and is able to translate that language into a standard language used by the operating system to access all disk drives. On UNIX, this is the language of block devices.

When the kernel has an appropriate device driver in place, it can then access the contents of the disk drive in raw format, which may contain one or more file systems. A file system driver is used to translate the commands used to access each specific file system into a standard set of commands that the operating system can use to talk to all file systems. Programs can then deal with these file systems on the basis of filenames, and directories/folders, contained within a hierarchical structure. They can create, delete, open, and close files, as well as gather various information about them, including access permissions, size, free space, and creation and modification dates.

Various differences between file systems make supporting all file systems difficult. Allowed characters in file names, case sensitivity, and the presence of various kinds of file attributes makes the implementation of a single interface for every file system a daunting task. Operating systems tend to recommend using (and so support natively) file systems specifically designed for them; for example, NTFS in Windows and ext3 and ReiserFS in Linux. However, in practice, third party drives are usually available to give support for the most widely used file systems in most general-purpose operating systems (for example, NTFS is available in Linux through NTFS-3g, and ext2/3 and ReiserFS are available in Windows through third-party software).

Support for file systems is highly varied among modern operating systems, although there are several common file systems which almost all operating systems include support and drivers for. Operating systems vary on file system support and on the disk formats they may be installed on. Under Windows, each file system is usually limited in application to certain media; for example, CDs must use ISO 9660 or UDF, and as of Windows Vista, NTFS is the only file system which the operating system can be installed on. It is possible to install Linux onto many types of file systems. Unlike other operating systems, Linux and UNIX allow any file system to be used regardless of the media it is stored in, whether it is a hard drive, a disc (CD, DVD...), a USB flash drive, or even contained within a file located on another file system.

Device drivers

Main article: Device driver

A device driver is a specific type of computer software developed to allow interaction with hardware devices. Typically this constitutes an interface for communicating with the device, through the specific computer bus or communications subsystem that the hardware is connected to, providing commands to and/or receiving data from the device, and on the other end, the requisite interfaces to the operating system and software applications. It is a specialized hardware-dependent computer program which is also operating system specific that enables another program, typically an operating system or applications software package or computer program running under the operating system kernel, to interact transparently with a hardware device, and usually provides the requisite interrupt handling necessary for any necessary asynchronous time-dependent hardware interfacing needs.

The key design goal of device drivers is abstraction. Every model of hardware (even within the same class of device) is different. Newer models also are released by manufacturers that provide more reliable or better performance and these newer models are often controlled differently. Computers and their operating systems cannot be expected to know how to control every device, both now and in the future. To solve this problem, operating systems essentially dictate how every type of device should be controlled. The function of the device driver is then to translate these operating system mandated function calls into device specific calls. In theory a new device, which is controlled in a new manner, should function correctly if a suitable driver is available. This new driver ensures that the device appears to operate as usual from the operating system's point of view.

Under versions of Windows before Vista and versions of Linux before 2.6, all driver execution was co-operative, meaning that if a driver entered an infinite loop it would freeze the system. More recent revisions of these operating systems incorporate kernel preemption, where the kernel interrupts the driver to give it tasks, and then separates itself from the process until it receives a response from the device driver, or gives it more tasks to do.

1.4.2 Networking

Main article: Computer network

Currently most operating systems support a variety of networking protocols, hardware, and applications for using them. This means that computers running dissimilar operating systems can participate in a common network for sharing resources such as computing, files, printers, and scanners using either wired or wireless connections. Networks can essentially allow a computer's operating system to access the resources of a remote computer to support the same functions as it could if those resources were connected directly to the local computer. This includes everything from simple communication, to using networked file systems or even sharing another computer's graphics or sound hardware. Some network services allow the resources of a computer to be accessed transparently, such as SSH which allows networked users direct access to a computer's command line interface.

Client/server networking allows a program on a computer, called a client, to connect via a network to another computer, called a server. Servers offer (or host) various services to other network computers and users. These services are usually provided through ports or numbered access points beyond the server's network address. Each port number is usually associated with a maximum of one running program, which is responsible for handling requests to that port. A daemon, being a user program, can in turn access the local hardware resources of that computer by passing requests to the operating system kernel.

Many operating systems support one or more vendor-specific or open networking protocols as well, for example, SNA on IBM systems, DECnet on systems from Digital Equipment Corporation, and Microsoft-specific protocols (SMB) on Windows. Specific protocols for specific tasks may also be supported such as NFS for file access. Protocols like ESound, or esd can be easily extended over the network to provide sound from local applications, on a remote system's sound hardware.

1.4.3 Security

Main article: Computer security

A computer being secure depends on a number of technologies working properly. A modern operating system provides access to a number of resources, which are available to software running on the system, and to external devices like networks via the kernel.

The operating system must be capable of distinguishing between requests which should be allowed to be processed, and others which should not be processed. While some systems may simply distinguish between "privileged" and "non-privileged", systems commonly have a form of requester *identity*, such as a user name. To establish identity there may be a process of *authentication*. Often a username must be quoted, and each username may have a password. Other methods of authentication, such as magnetic cards or biometric data, might be used instead. In some cases, especially connections from the network, resources may be accessed with no authentication at all (such as reading files over a network share). Also covered by the concept of requester **identity** is *authorization*; the particular services and resources accessible by the requester once logged into a system are tied to either the requester's user account or to the variously configured groups of users to which the requester belongs.

In addition to the allow or disallow model of security, a system with a high level of security also offers auditing options. These would allow tracking of requests for access to resources (such as, "who has been reading this file?"). Internal security, or security from an already running program is only possible if all possibly harmful requests must be carried out through interrupts to the operating system kernel. If programs can directly access hardware and resources, they cannot be secured.

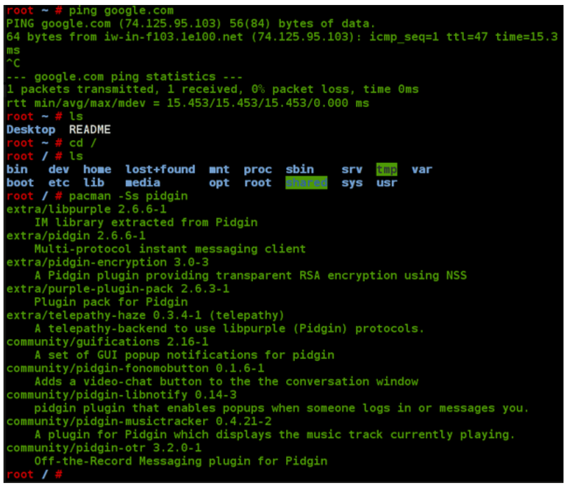
External security involves a request from outside the computer, such as a login at a connected console or some kind of network connection. External requests are often passed through device drivers to the operating system's kernel, where they can be passed onto applications, or carried out directly. Security of operating systems has long been a concern because of highly sensitive data held on computers, both of a commercial and military nature. The United States Government Department of Defense (DoD) created the *Trusted Computer System Evaluation Criteria* (TCSEC) which is a standard that sets basic requirements for assessing the effectiveness of security. This became of vital importance to operating system makers, because the TCSEC was used to evaluate, classify and select trusted operating systems being considered for the processing, storage and retrieval of sensitive or classified information.

Network services include offerings such as file sharing, print services, email, web sites, and file transfer protocols (FTP), most of which can have compromised security. At the front line of security are hardware devices known as firewalls or intrusion detection/prevention systems. At the operating system level, there are a number of software firewalls available, as well as intrusion detection/prevention systems. Most modern operating systems include a software firewall, which is enabled by default. A software firewall can be configured to allow or deny network traffic to or from a service or application running on the operating system. Therefore, one can install and be running an insecure service, such as Telnet or FTP, and not have to be threatened by a security breach because the firewall would deny all traffic trying to connect to the service on that port.

An alternative strategy, and the only sandbox strategy available in systems that do not meet the Popek and Goldberg virtualization requirements, is where the operating system is not running user programs as native code, but instead either emulates a processor or provides a host for a p-code based system such as Java.

Internal security is especially relevant for multi-user systems; it allows each user of the system to have private files that the other users cannot tamper with or read. Internal security is also vital if auditing is to be of any use, since a program can potentially bypass the operating system, inclusive of bypassing auditing.

1.4.4 User interface



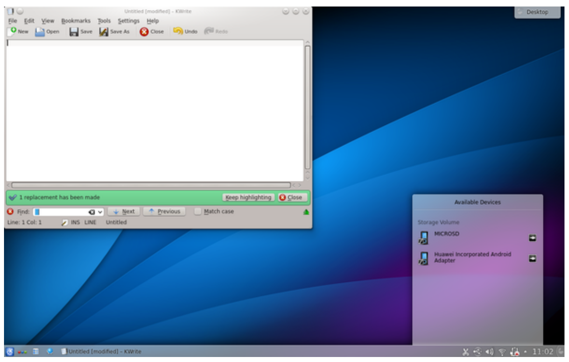
```
root ~ # ping google.com
PING google.com (74.125.95.103) 56(84) bytes of data.
64 bytes from iw-in-f103.1e100.net (74.125.95.103): icmp_seq=1 ttl=47 time=15.3
ms
^C
--- google.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 15.453/15.453/15.453/0.000 ms
root ~ # ls
Desktop  README
root ~ # cd /
root / # ls
bin   dev  home  lost+found  mnt  proc  sbin    srv  tmp  var
boot  etc  lib   media       opt  root  shared  sys  usr
root / # pacman -Ss pidgin
extra/libpurple 2.6.6-1
    IM library extracted from Pidgin
extra/pidgin 2.6.6-1
    Multi-protocol instant messaging client
extra/pidgin-encryption 3.0-3
    A Pidgin plugin providing transparent RSA encryption using NSS
extra/purple-plugin-pack 2.6.3-1
    Plugin pack for Pidgin
extra/telepathy-haze 0.3.4-1 (telepathy)
    A telepathy-backend to use libpurple (Pidgin) protocols.
community/guifications 2.16-1
    A set of GUI popup notifications for pidgin
community/pidgin-fonomobutton 0.1.6-1
    Adds a video-chat button to the the conversation window
community/pidgin-libnotify 0.14-3
    pidgin plugin that enables popups when someone logs in or messages you.
community/pidgin-musictracker 0.4.21-2
    A plugin for Pidgin which displays the music track currently playing.
community/pidgin-otr 3.2.0-1
    Off-the-Record Messaging plugin for Pidgin
root / #
```

A screenshot of the Bourne Again Shell command line. Each command is typed out after the 'prompt', and then its output appears below, working its way down the screen. The current command prompt is at the bottom.

Main article: Operating system user interface

Every computer that is to be operated by an individual requires a user interface. The user interface is usually referred to as a shell and is essential if human interaction is to be supported. The user interface views the directory structure and requests services from the operating system that will acquire data from input hardware devices, such as a keyboard, mouse or credit card reader, and requests operating system services to display prompts, status messages and such on output hardware devices, such as a video monitor or printer. The two most common forms of a user interface have historically been the command-line interface, where computer commands are typed out line-by-line, and the graphical user interface, where a visual environment (most commonly a WIMP) is present.

Graphical user interfaces



A screenshot of the KDE Plasma Desktop graphical user interface. Programs take the form of images on the screen, and the files, folders (directories), and applications take the form of icons and symbols. A mouse is used to navigate the computer.

Most of the modern computer systems support graphical user interfaces (GUI), and often include them. In some computer systems, such as the original implementation of Mac OS, the GUI is integrated into the kernel.

While technically a graphical user interface is not an operating system service, incorporating support for one into the operating system kernel can allow the GUI to be more responsive by reducing the number of context switches required for the GUI to perform its output functions. Other operating systems are modular, separating the graphics subsystem from the kernel and the Operating System. In the 1980s UNIX, VMS and many others had operating systems that were built this way. Linux and Mac OS X are also built this way. Modern releases of Microsoft Windows such as Windows Vista implement a graphics subsystem that is mostly in user-space; however the graphics drawing routines of versions between Windows NT 4.0 and Windows Server 2003 exist mostly in kernel space. Windows 9x had very little distinction between the interface and the kernel.

Many computer operating systems allow the user to install or create any user interface they desire. The X Window System in conjunction with GNOME or KDE Plasma Desktop is a commonly found setup on most Unix and Unix-like (BSD, Linux, Solaris) systems. A number of Windows shell replacements have been released for Microsoft Windows, which offer alternatives to the included Windows shell, but the shell itself cannot be separated from Windows.

Numerous Unix-based GUIs have existed over time, most derived from X11. Competition among the various vendors of Unix (HP, IBM, Sun) led to much fragmentation, though an effort to standardize in the 1990s to COSE and CDE failed for various reasons, and were eventually eclipsed by the widespread adoption of GNOME and K Desktop Environment. Prior to free software-based toolkits and desktop environments, Motif was the prevalent toolkit/desktop combination (and was the basis upon which CDE was developed).

Graphical user interfaces evolve over time. For example, Windows has modified its user interface almost every time a new major version of Windows is released, and the Mac OS GUI changed dramatically with the introduction of Mac OS X in 1999.[31]

1.5 Real-time operating systems

Main article: Real-time operating system

A real-time operating system (RTOS) is an operating system intended for applications with fixed deadlines (real-time computing). Such applications include some small embedded systems, automobile engine controllers, industrial robots, spacecraft, industrial control, and some large-scale computing systems.

An early example of a large-scale real-time operating system was Transaction Processing Facility developed by American Airlines and IBM for the Sabre Airline Reservations System.

Embedded systems that have fixed deadlines use a real-time operating system such as VxWorks, PikeOS, eCos, QNX, MontaVista Linux and RTLinux. Windows CE is a real-time operating system that shares similar APIs to desktop Windows but shares none of desktop Windows' codebase. Symbian OS also has an RTOS kernel (EKA2) starting with version 8.0b.

Some embedded systems use operating systems such as Palm OS, BSD, and Linux, although such operating systems do not support real-time computing.

1.6 Operating system development as a hobby

See also: Hobbyist operating system development

Operating system development is one of the most complicated activities in which a computing hobbyist may engage. A hobby operating system may be classified as one whose code has not been directly derived from an existing operating system, and has few users and active developers.[32]

In some cases, hobby development is in support of a "homebrew" computing device, for example, a simple single-board computer powered by a 6502 microprocessor. Or, development may be for an architecture already in widespread use. Operating system development may come from entirely new concepts, or may commence by modeling an existing operating system. In either case, the hobbyist is his/her own developer, or may interact with a small and sometimes unstructured group of individuals who have like interests.

Examples of a hobby operating system include ReactOS and Syllable.

1.7 Diversity of operating systems and portability

Application software is generally written for use on a specific operating system, and sometimes even for specific hardware. When porting the application to run on another OS, the functionality required by that application may be implemented differently by that OS (the names of functions, meaning of arguments, etc.) requiring the application to be adapted, changed, or otherwise maintained.

Unix was the first operating system not written in assembly language, making it very portable to systems different from its native PDP-11.[33]

This cost in supporting operating systems diversity can be avoided by instead writing applications against software platforms like Java or Qt. These abstractions have already borne the cost of adaptation to specific operating systems and their system libraries.

Another approach is for operating system vendors to adopt standards. For example, POSIX and OS abstraction layers provide commonalities that reduce porting costs.

1.8 Market share

Main article: Usage share of operating systems

Source: Gartner

1.9 See also

1.10 References

[1] Stallings (2005). *Operating Systems, Internals and Design Principles*. Pearson: Prentice Hall. p. 6.

[2] Dhotre, I.A. (2009). *Operating Systems*. Technical Publications. p. 1.

[3] “Operating System Market Share”. Net Applications.

[4] Silberschatz Galvin Gagne (2012). *Operating Systems Concepts*. New York: Wiley. ISBN 978-1118063330.

[5] Hansen, Per Brinch, ed. (2001). *Classic Operating Systems*. Springer. pp. 4–7. ISBN 0-387-95113-X.

[6] Lavington 1980, pp. 50—52

[7] Brinch Hansen 2000

[8] Ritchie, Dennis. “Unix Manual, first edition”. Lucent Technologies. Retrieved 22 November 2012.

[9] “Apple introduces mac OS X Maverick’s at WWDC”. *YouTube*. TechandPlayTV. June 10, 2013. Retrieved November 17, 2013.

[10] “OS X Mountain Lion – Move your Mac even further ahead”. Apple. Retrieved 2012-08-07.

[11] Usage share of operating systems

[12] “Top 5 Operating Systems from January to April 2011”. StatCounter. October 2009. Retrieved November 5, 2009.

[13] “IDC report into Server market share”. Idc.com. Retrieved 2012-08-07.

[14] “Linux still top embedded OS”. Archived from the original on 2012-05-29.

[15] Jermoluk, Tom (2012-08-03). “TOP500 List – November 2010 (1–100) | TOP500 Supercomputing Sites”. Top500.org. Retrieved 2012-08-07.

[16] “Chromium OS”. Chromium.org.

[17] “Chromium OS FAQ”. The Chromium Projects. Retrieved 28 February 2014.

[18] “Global Web Stats”. Net Market Share, Net Applications. May 2011. Retrieved 2011-05-07.

[19] “Global Web Stats”. W3Counter, Awio Web Services. September 2009. Retrieved 2009-10-24.

[20] “Operating System Market Share”. Net Applications. October 2009. Retrieved November 5, 2009.

[21] “w3schools.com OS Platform Statistics”. Retrieved October 30, 2011.

[22] "Stats Count Global Stats Top Five Operating Systems". Retrieved October 30, 2011.

[23] "Global statistics at w3counter.com". Retrieved 23 January 2012.

[24] "Troubleshooting MS-DOS Compatibility Mode on Hard Disks". Support.microsoft.com. Retrieved 2012-08-07.

[25] "Using NDIS 2 PCMCIA Network Card Drivers in Windows 95". Support.microsoft.com. Retrieved 2012-08-07.

[26] "INFO: Windows 95 Multimedia Wave Device Drivers Must be 16 bit". Support.microsoft.com. Retrieved 2012-08-07.

[27] Arthur, Charles. "Windows 8 will run on ARM chips - but third-party apps will need rewrite". *The Guardian*.

[28] "Operating System Share by Groups for Sites in All Locations January 2009".

[29] "Behind the IDC data: Windows still No. 1 in server operating systems". ZDNet. 2010-02-26.

[30] Stallings, William (2008). *Computer Organization & Architecture*. New Delhi: Prentice-Hall of India Private Limited. p. 267. ISBN 978-81-203-2962-1.

[31] Poisson, Ken. "Chronology of Personal Computer Software". Retrieved on 2008-05-07. Last checked on 2009-03-30.

[32] "My OS is less hobby than yours". *Osnews*. December 21, 2009. Retrieved December 21, 2009.

[33] "The History of Unix". *BYTE*. August 1983. p. 188. Retrieved 31 January 2015.

[34] Lance Whitney (January 7, 2014). "Android device shipments to top 1 billion this year -- Gartner".

1.11 Further reading

- Auslander, Marc A.; Larkin, David C.; Scherr, Allan L. (1981). "The evolution of the MVS Operating System". IBM J. Research & Development.
- Deitel, Harvey M.; Deitel, Paul; Choffnes, David. *Operating Systems*. Pearson/Prentice Hall. ISBN 978-0-13-092641-8.
- Bic, Lubomur F.; Shaw, Alan C. (2003). *Operating Systems*. Pearson: Prentice Hall.
- Silberschatz, Avi; Galvin, Peter; Gagne, Greg (2008). *Operating Systems Concepts*. John Wiley & Sons. ISBN 0-470-12872-0.
- O'Brien, J.A., & Marakas, G.M.(2011). Management Information Systems. 10e. McGraw-Hill Irwin
- Leva, Alberto; Maggio, Martina; Papadopoulos, Alessandro Vittorio; Terraneo, Federico (2013). *Control-based Operating System Design*. IET. ISBN 978-1-84919-609-3.

1.12 External links

- Operating Systems at DMOZ
- Multics History and the history of operating systems

Chapter 2

System software

Not to be confused with Software system.

System software (**systems software**) is computer software designed to operate and control the computer hardware and to provide a platform for running application software.[1] System software can be separated into two different categories, operating systems and utility software.

- The *operating system* (prominent examples being z/OS, Microsoft Windows, Mac OS X and Linux), allows the parts of a computer to work together by performing tasks like transferring data between memory and disks or rendering output onto a display device. It also provides a platform to run high-level system software and application software.
  - A *kernel* is the core part of the operating system that defines an API for applications programs (including some system software) and an interface to device drivers.
    - Device drivers such as computer BIOS and device firmware provide basic functionality to operate and control the hardware connected to or built into the computer.
  - A *user interface* "allows users to interact with a computer."[2] Since the 1980s the graphical user interface (GUI) has been perhaps the most common user interface technology. The command-line interface is still a commonly used alternative.
- *Utility software* helps to analyze, configure, optimize and maintain the computer, such as virus protection.[3]

In some publications, the term *system software* also includes software development tools (like a compiler, linker or debugger).[4]

In contrast to system software, software that allows users to do things like create text documents, play games, listen to music, or web browsers to surf the web are called application software.[5] The line where the distinction should be drawn isn't always clear. Most operating systems bundle such software. Such software is not considered *system software* when it can be uninstalled without affecting the functioning of other software. Exceptions could be e.g. web browsers such as Internet Explorer where Microsoft argued in court that it was system software that could not be uninstalled. Later examples are Chrome OS and Firefox OS where the browser functions as the only user interface *and* the only way to run programs (and other web browser can not be installed in their place), then they can well be argued to *be* (part of) the operating system and then system software.

2.1 See also

- Systems programming language

2.2 References

[1] "What is software? - Definition from WhatIs.com". Searchsoa.techtarget.com. Retrieved 2012-06-24.

[2] Daeryong, Kim. "Microcomputer Information Technology". Retrieved 2013-09-22.

[3] "What is System Software?". Alverno.edu. 2011-07-24. Archived from the original on 2011-07-24. Retrieved 2012-06-24.

[4] "What is systems software? - A Word Definition From the Webopedia Computer Dictionary". Webopedia.com. Retrieved 2012-06-24.

[5] W. W. Milner, Ann Montgomery-Smith (2000). *Information and Communication technology for Intermediate Gnvq*. p.126

2.3 External links

- Sammet, Jean (October 1971). "Brief Survey of Languages Used for Systems Implementation". *SCM SIGPLAN Notices* **6** (9): 1–19. doi:10.1145/942596.807055.

Chapter 3

Firmware

A typical firmware-controlled device: a television remote control. Consumer products like this have been using firmware since the 1970s.

In electronic systems and computing, **firmware** is "the combination of a hardware device, e.g. an integrated circuit, and computer instructions and data that reside as read only software on that device". As a result, firmware usually cannot be modified during normal operation of the device.[1] Typical examples of devices containing firmware are embedded systems (such as traffic lights, consumer appliances, and digital watches), computers, computer peripherals, mobile phones, and digital cameras. The firmware contained in these devices provides the control program for the device.

Firmware is held in non-volatile memory devices such as ROM, EPROM, or flash memory. Changing the firmware of a device may rarely or never be done during its economic lifetime; some firmware memory devices are permanently installed and cannot be changed after manufacture. Common reasons for updating firmware include fixing bugs or adding features to the device. This may require ROM integrated circuits to be physically replaced, or flash memory to be reprogrammed through a special procedure.[2] Firmware such as the ROM BIOS of a personal computer may contain only elementary basic functions of a device and may only provide services to higher-level software. Firmware such as the program of an embedded system may be the only program that will run on the system and provide all of its functions.

Before integrated circuits, other firmware devices included a discrete semiconductor diode matrix. The Apollo guidance computer had firmware consisting of a specially manufactured core memory plane, called "core rope memory", where data were stored by physically threading wires through (1) or around (0) the core storing each data bit.[3]

3.1 Origin of the term

Ascher Opler coined the term "firmware" in a 1967 *Datamation* article.[4] Originally, it meant the contents of a writable control store (a small specialized high speed memory), containing microcode that defined and implemented the computer's instruction set, and that could be reloaded to specialize or modify the instructions that the central processing unit (CPU) could execute. As originally used, firmware contrasted with hardware (the CPU itself) and software (normal instructions executing on a CPU). It was not composed of CPU machine instructions, but of lower-level microcode involved in the implementation of machine instructions. It existed on the boundary between hardware and software; thus the name "firmware".

Still later, popular usage extended the word "firmware" to denote anything ROM-resident, including processor machine-instructions for BIOS, bootstrap loaders, or specialized applications.

Until the mid-1990s, updating firmware typically involved replacing a storage medium containing firmware, usually a socketed ROM integrated circuit. Flash memory allows firmware to be updated without physically removing an integrated circuit from the system. An error during the update process may make the device non-functional, or "bricked".

3.2 Personal computers

In some respects, the various firmware components are as important as the operating system in a working computer. However, unlike most modern operating systems,

ROM BIOS firmware on a Baby AT motherboard

firmware rarely has a well-evolved automatic mechanism of updating itself to fix any functionality issues detected after shipping the unit.

The BIOS may be “manually” updated by a user, using a small utility program. In contrast, firmware in storage devices (harddisks, DVD drives, flash storage) rarely gets updated, even when flash (rather than ROM) storage is used for the firmware; there are no standardized mechanisms for detecting or updating firmware versions.

Most computer peripherals are themselves special-purpose computers. Devices such as printers, scanners, cameras, USB drives, have firmware stored internally. Some devices may permit field replacement of firmware.

Some low-cost peripherals no longer contain non-volatile memory for firmware, and instead rely on the host system to transfer the device control program from a disk file or CD.[5]

3.3 Consumer products

As of 2010 most portable music players support firmware upgrades. Some companies use firmware updates to add new playable file formats (codecs); iriver added Vorbis playback support this way, for instance. Other features that may change with firmware updates include the GUI or even the battery life. Most mobile phones have a Firmware Over The Air firmware upgrade capability for much the same reasons; some may even be upgraded to enhance reception or sound quality, illustrating the fact that firmware is used at more than one level in complex products (in a CPU-like microcontroller versus in a digital signal processor, in this particular case).

3.4 Automobiles

Since 1996 most automobiles have employed an on-board computer and various sensors to detect mechanical problems. As of 2010 modern vehicles also employ computer-controlled ABS systems and computer-operated Transmission Control Units (TCU). The driver can also get in-dash information while driving in this manner, such as real-time fuel-economy and tire-pressure readings. Local dealers can update most vehicle firmware.

3.5 Examples

Examples of firmware include:

- In consumer products:
  - Timing and control systems for washing machines
  - Controlling sound and video attributes, as well as the channel list, in modern TVs
  - EPROM chips used in the Eventide H-3000 series of digital music processors
- In computers:
  - The BIOS found in IBM-compatible personal computers
  - The (U)EFI-compliant firmware used on Itanium systems, Intel-based computers from Apple, and many Intel desktop computer motherboards
  - Open Firmware, used in SPARC-based computers from Sun Microsystems and Oracle Corporation, PowerPC-based computers from Apple, and computers from Genesi
  - ARCS, used in computers from Silicon Graphics
  - Kickstart, used in the Amiga line of computers (POST, hardware init + Plug and Play auto-configuration of peripherals, kernel, etc.)
  - RTAS (Run-Time Abstraction Services), used in computers from IBM
  - The Common Firmware Environment (CFE)
- In routers and firewalls:
  - OpenWrt – an open-source firewall/router OS based on Linux
  - m0n0wall – an embedded firewall distribution of FreeBSD
  - IPFire – a free Linux router and firewall distribution
  - fli4l – a free Linux router and firewall distribution
- In NAS systems:
  - NAS4Free – an open-source NAS operating system based on FreeBSD 9.1
  - Openfiler – a free Linux-based NAS operating system

3.6 Flashing

Flashing[6] involves the overwriting of existing firmware or data on EEPROM modules present in an electronic device with new data.[6] This can be done to upgrade a device[7] or to change the provider of a service associated with the function of the device, such as changing from one mobile phone service provider to another or installing a new operating system. If firmware is upgradable, it is often done via a program from the provider, and will often allow the old firmware to be saved before upgrading so it can be reverted to if the process fails, or if the newer version performs worse.

3.7 Firmware hacking

Sometimes third parties create an unofficial new or modified ("aftermarket") version of firmware to provide new features or to unlock hidden functionality. Examples include:

- Rockbox for digital audio players.
- CHDK[8] and Magic Lantern[8] for Canon digital cameras.
- Nikon Hacker project for Nikon EXPEED DSLRs.
- Many third-party firmware projects for wireless routers, including:
    - OpenWrt, and its derivatives such as DD-WRT, for wireless routers.[8]
    - RouterTech – for ADSL modem/routers based on the Texas Instruments AR7 chipset (with the Pspboot or Adam2 bootloader).
    - List of wireless router firmware projects
- Firmware that allows DVD drives to be region-free.
- SamyGO, modified firmware for Samsung televisions.[9]
- Many homebrew projects for gaming consoles. These often unlock general-purpose computing functionality in previously limited devices (e.g., running Doom on iPods).

Most firmware hacks are free and open source software as well.

These hacks usually take advantage of the firmware update facility on many devices to install or run themselves. Some, however, must resort to exploits in order to run, because the manufacturer has attempted to lock the hardware to stop it from running unlicensed code.

3.7.1 HDD firmware hacks

The Moscow-based Kaspersky Lab discovered that a group of developers it refers to as the "Equation Group" has developed hard disk drive firmware modifications for various drive models, containing a trojan horse that allows data to be stored on the drive in locations that will not be erased even if the drive is formatted or wiped.[10] Although the Kaspersky Lab report did not explicitly claim that this group is part of the United States National Security Agency (NSA), evidence obtained from the code of various Equation Group software suggests that they are part of the NSA.[11][12]

Researchers from the Kaspersky Lab categorized the undertakings by Equation Group as the most advanced hacking operation ever uncovered, also documenting around 500 infections caused by the Equation Group in at least 42 countries.

3.8 Security risks

Mark Shuttleworth, founder of the Ubuntu Linux distribution, has described proprietary firmware as a security risk,[13] saying that "firmware on your device is the NSA's best friend" and calling firmware "a trojan horse of monumental proportions". He has pointed out that low-quality, closed source firmware is a major threat to system security:[14] "Your biggest mistake is to assume that the NSA is the only institution abusing this position of trust – in fact, it's reasonable to assume that all firmware is a cesspool of insecurity, courtesy of incompetence of the highest degree from manufacturers, and competence of the highest degree from a very wide range of such agencies".

As a solution to this problem, he has called for declarative firmware.[14] Firmware should be open source so that the code can be checked and verified; it should also be declarative, meaning that it should describe "hardware linkage and dependencies" and "should not include executable code".[14]

Custom firmware hacks have also focused on injecting malware into devices such as smartphones or USB devices. One such smartphone injection was demonstrated on the Symbian OS at MalCon,[15][16] a hacker convention. A USB device firmware hack called *BadUSB* was presented at Black Hat USA 2014 conference,[17] demonstrating how a USB flash drive microcontroller can be reprogrammed to spoof various other device types in order to take control of a computer, exfiltrate data, or spy on the user.[18][19] Other security researchers have worked further on how to exploit the principles behind BadUSB,[20] releasing at the same time the source code of hacking tools that can be used to modify the behavior of USB flash drives.[21]

3.9 See also

- ROM image
- UEFI
- Coreboot
- Microcode
- Binary blob
- Bootloader
- Aftermarket firmware category

3.10 References

[1] "Glossary of Computer System Software Development Terminology (8/95)". *fda.gov*. FDA. November 25, 2014. Retrieved March 1, 2015.

[2] "What is firmware?". incepator.pinzaru.ro. Retrieved 2013-06-14.

[3] Dag Spicer (August 12, 2000). "One Giant Leap: The Apollo Guidance Computer". Dr. Dobbs. Retrieved August 24, 2012.

[4] Opler, Ascher (January 1967). "Fourth-Generation Software". *Datamation* **13** (1): 22–24.

[5] Corbet, Jonathan; Rubini, Alessandro; Kroah-Hartman, Greg (2005). *Linux Device Drivers*. O'Reilly Media. p. 405. ISBN 0596005903.

[6] "Flashing Firmware". Tech-Faq.com. Retrieved July 8, 2011.

[7] "HTC Developer Center". HTC. Archived from the original on April 26, 2011. Retrieved July 8, 2011.

[8] "Custom Firmware Rocks!". 2009-08-05. Retrieved 2009-08-13.

[9] "SamyGO: replacing television firmware". LWN.net. 2009-11-14. Retrieved 2009-12-11.

[10] "Equation Group: The Crown Creator of Cyber-Espionage". Kaspersky Lab. February 16, 2015.

[11] Dan Goodin (February 2015). "How "omnipotent" hackers tied to NSA hid for 14 years—and were found at last". *Ars Technica*.

[12] "Breaking: Kaspersky Exposes NSA's Worldwide, Backdoor Hacking of Virtually All Hard-Drive Firmware". Daily Kos. February 17, 2015.

[13] Linux Format n°184, June 2014, page 7.

[14] Linux Magazine issue 162, May 2014, page 9.

[15] "We will be back soon!". Malcon.org. Retrieved 2013-06-14.

[16] "Hacker plants back door in Symbian firmware". H-online.com. 2010-12-08. Archived from the original on 21 May 2013. Retrieved 2013-06-14.

[17] "Why the Security of USB Is Fundamentally Broken". Wired.com. 2014-07-31. Retrieved 2014-08-04.

[18] "BadUSB - On Accessories that Turn Evil". BlackHat.com. Retrieved 2014-08-06.

[19] Karsten Nohl; Sascha Krißler; Jakob Lell (2014-08-07). "BadUSB – On accessories that turn evil" (PDF). *srlabs.de*. Retrieved 2014-08-23.

[20] "BadUSB Malware Released - Infect millions of USB Drives". *The Hacking Post - Latest hacking News & Security Updates*. Retrieved 7 October 2014.

[21] "The Unpatchable Malware That Infects USBs Is Now on the Loose". *WIRED*. Retrieved 7 October 2014.

3.11 External links

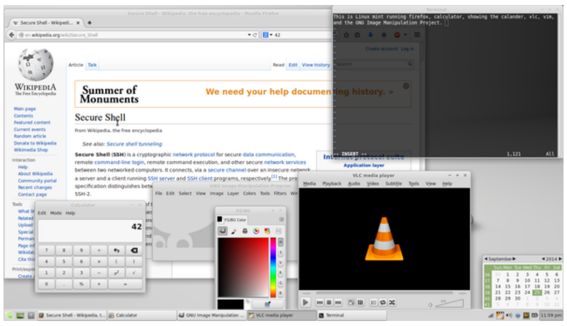
- BadUSB - On Accessories that Turn Evil on YouTube, by Karsten Nohl and Jakob Lell
- Phison 2251-03 (2303) Custom Firmware & Existing Firmware Patches (BadUSB)
- Hard disk hacking (includes an analysis of feasible security exploits through firmware modifications, in eight parts)
- Snake on a keyboard (firmware modifications, in seven parts)

Chapter 4

Computer multitasking

For other uses, see Multitasking (disambiguation).

In computing, **multitasking** is a method where multiple



Modern desktop operating systems are capable of handling large numbers of different processes at the same time. This screenshot shows Linux Mint running simultaneously Xfce desktop environment, Firefox, a calculator program, the built-in calendar, Vim, GIMP, and VLC media player.

tasks (also known as processes) are performed during the same period of time – they are executed concurrently (in overlapping time periods, new tasks starting before others have ended) instead of sequentially (one completing before the next starts). The tasks share common processing resources, such as central processing units (CPUs) and main memory.

Multitasking does *not* necessarily mean that multiple tasks are executing at exactly the same instant. In other words, multitasking does not imply parallel execution, but it does mean that more than one task can be part-way through execution at the same time, and that more than one task is advancing over a given period of time.

In the case of a computer with a single CPU, only one task is said to be running at any point in time, meaning that the CPU is actively executing instructions for that task. Multitasking solves the problem by scheduling which task may be the one running at any given time, and when another waiting task gets a turn. The act of reassigning a CPU from one task to another one is called a context switch. When context switches occur frequently enough, the illusion of parallelism is achieved.

Even on multiprocessor or multicore computers, which have multiple CPUs/cores so more than one task can be executed at once (physically, one per CPU or core), multitasking allows many more tasks to be run than there are CPUs. The term *multitasking* has become an international term, as the same word is used in many other languages such as German, Italian, Dutch, Danish and Norwegian.

Operating systems may adopt one of many different scheduling strategies, which generally fall into the following categories:

- In multiprogramming systems, the running task keeps running until it performs an operation that requires waiting for an external event (e.g. reading from a tape) or until the computer's scheduler forcibly swaps the running task out of the CPU. Multiprogramming systems are designed to maximize CPU usage.
- In time-sharing systems, the running task is required to relinquish the CPU, either voluntarily or by an external event such as a hardware interrupt. Time sharing systems are designed to allow several programs to execute apparently simultaneously.
- In real-time systems, some waiting tasks are guaranteed to be given the CPU when an external event occurs. Real time systems are designed to control mechanical devices such as industrial robots, which require timely processing.

4.1 Multiprogramming

In the early days of computing, CPU time was expensive, and peripherals were very slow. When the computer ran a program that needed access to a peripheral, the central processing unit (CPU) would have to stop executing program instructions while the peripheral processed the data. This was deemed very inefficient.

The first computer using a multiprogramming system was the British *Leo III* owned by J. Lyons and Co. Several different programs in batch were loaded in the computer memory, and the first one began to run. When the first program reached an instruction waiting for a peripheral, the context of this program was stored away, and the second program in memory was given a chance to run. The process continued until all programs finished running.

The use of multiprogramming was enhanced by the arrival of virtual memory and virtual machine technology, which enabled individual programs to make use of memory and operating system resources as if other concurrently running programs were, for all practical purposes, non-existent and invisible to them.

Multiprogramming doesn't give any guarantee that a program will run in a timely manner. Indeed, the very first program may very well run for hours without needing access to a peripheral. As there were no users waiting at an interactive terminal, this was no problem: users handed in a deck of punched cards to an operator, and came back a few hours later for printed results. Multiprogramming greatly reduced wait times when multiple batches were being processed.

4.2 Cooperative multitasking

See also: Nonpreemptive multitasking

The expression “time sharing” usually designated computers shared by interactive users at terminals, such as IBM’s TSO, and VM/CMS. The term “time-sharing” is no longer commonly used, having been replaced by “multitasking”, following the advent of personal computers and workstations rather than shared interactive systems.

Early multitasking systems used applications that voluntarily ceded time to one another. This approach, which was eventually supported by many computer operating systems, is known today as cooperative multitasking. Although it is now rarely used in larger systems except for specific applications such as CICS or the JES2 subsystem, cooperative multitasking was once the scheduling scheme employed by Microsoft Windows (prior to Windows 95 and Windows NT) and Mac OS (prior to OS X) in order to enable multiple applications to be run simultaneously. Windows 9x also used cooperative multitasking, but only for 16-bit legacy applications, much the same way as pre-Leopard PowerPC versions of Mac OS X used it for Classic applications. The network operating system NetWare used cooperative multitasking up to NetWare 6.5. Cooperative multitasking is still used today on RISC OS systems.[1]

As a cooperatively multitasked system relies on each process regularly giving up time to other processes on the system, one poorly designed program can consume all of the CPU time for itself, either by performing extensive calculations or by busy waiting; both would cause the whole system to hang. In a server environment, this is a hazard that makes the entire environment unacceptably fragile.

4.3 Preemptive multitasking

Main article: Preemption (computing)

Preemptive multitasking allows the computer system to guarantee more reliably each process a regular “slice” of operating time. It also allows the system to deal rapidly with important external events like incoming data, which might require the immediate attention of one or another process. Operating systems were developed to take advantage of these hardware capabilities and run multiple processes preemptively. Preemptive multitasking was supported on DEC’s PDP-8 computers, and implemented in OS/360 MFT in 1967, in MULTICS (1964), and Unix (1969); it is a core feature of all Unix-like operating systems, such as Linux, Solaris and BSD with its derivatives.[2]

At any specific time, processes can be grouped into two categories: those that are waiting for input or output (called "I/O bound"), and those that are fully utilizing the CPU ("CPU bound"). In primitive systems, the software would often "poll", or "busywait" while waiting for requested input (such as disk, keyboard or network input). During this time, the system was not performing useful work. With the advent of interrupts and preemptive multitasking, I/O bound processes could be “blocked”, or put on hold, pending the arrival of the necessary data, allowing other processes to utilize the CPU. As the arrival of the requested data would generate an interrupt, blocked processes could be guaranteed a timely return to execution.

The earliest preemptive multitasking OS available to home users was Sinclair QDOS on the Sinclair QL, released in 1984, but very few people bought the machine. Commodore’s powerful Amiga, released the following year, was the first commercially successful home computer to use the technology, and its multimedia abilities make it a clear ancestor of contemporary multitasking personal computers. Microsoft made preemptive multitasking a core feature of their flagship operating system in the early 1990s when developing Windows NT 3.1 and then Windows 95. It was later adopted on the Apple Macintosh by Mac OS X that, as a Unix-like operating system, uses preemptive multitasking for all native applications.

A similar model is used in Windows 9x and the Windows NT family, where native 32-bit applications are multitasked preemptively, and legacy 16-bit Windows 3.x programs are multitasked cooperatively within a single process, although in the NT family it is possible to force a 16-bit application to run as a separate preemptively multitasked process.[3] 64-bit editions of Windows, both for the x86-64 and Itanium architectures, no longer provide support for legacy 16-bit applications, and thus provide preemptive multitasking for all supported applications.

4.4 Real time

Another reason for multitasking was in the design of real-time computing systems, where there are a number of possibly unrelated external activities needed to be controlled by a single processor system. In such systems a hierarchical interrupt system is coupled with process prioritization to ensure that key activities were given a greater share of available process time.

4.5 Multithreading

As multitasking greatly improved the throughput of computers, programmers started to implement applications as sets of cooperating processes (e. g., one process gathering input data, one process processing input data, one process writing out results on disk). This, however, required some tools to allow processes to efficiently exchange data.

Threads were born from the idea that the most efficient way for cooperating processes to exchange data would be to share their entire memory space. Thus, threads are effectively processes that run in the same memory context and share other resources with their parent processes, such as open files. Threads are described as *lightweight processes* because switching between threads does not involve changing the memory context.[4][5][6]

While threads are scheduled preemptively, some operating systems provide a variant to threads, named *fibers*, that are scheduled cooperatively. On operating systems that do not provide fibers, an application may implement its own fibers using repeated calls to worker functions. Fibers are even more lightweight than threads, and somewhat easier to program with, although they tend to lose some or all of the benefits of threads on machines with multiple processors.

Some systems directly support multithreading in hardware.

4.6 Memory protection

Main article: Memory protection

Essential to any multitasking system is to safely and effectively share access to system resources. Access to memory must be strictly managed to ensure that no process can inadvertently or deliberately read or write to memory locations outside of the process's address space. This is done for the purpose of general system stability and data integrity, as well as data security.

In general, memory access management is the operating system kernel's responsibility, in combination with hardware mechanisms (such as the memory management unit (MMU)) that provide supporting functionalities. If a process attempts to access a memory location outside of its memory space, the MMU denies the request and signals the kernel to take appropriate actions; this usually results in forcibly terminating the offending process. Depending on the software and kernel design and the specific error in question, the user may receive an access violation error message such as “segmentation fault”.

In a well designed and correctly implemented multitasking system, a given process can never directly access memory that belongs to another process. An exception to this rule is in the case of shared memory; for example, in the System V inter-process communication mechanism the kernel allocates memory to be mutually shared by multiple processes. Such features are often used by database management software such as PostgreSQL.

Inadequate memory protection mechanisms, either due to flaws in their design or poor implementations, allow for security vulnerabilities that may be potentially exploited by malicious software.

4.7 Memory swapping

Use of a swap file or swap partition is a way for the operating system to provide more memory than is physically available by keeping portions of the primary memory in secondary storage. While multitasking and memory swapping are two completely unrelated techniques, they are very often used together, as swapping memory allows more tasks to be loaded at the same time. Typically, a multitasking system allows another process to run when the running process hits a point where it has to wait for some portion of memory to be reloaded from secondary storage.

4.8 Programming

Processes that are entirely independent are not much trouble to program in a multitasking environment. Most of the complexity in multitasking systems comes from the need to share computer resources between tasks and to synchronize the operation of co-operating tasks.

Various concurrent computing techniques are used to avoid potential problems caused by multiple tasks attempting to access the same resource.

Bigger systems were sometimes built with a central processor(s) and some number of I/O processors, a kind of asymmetric multiprocessing.

Over the years, multitasking systems have been refined. Modern operating systems generally include detailed mechanisms for prioritizing processes, while symmetric multiprocessing has introduced new complexities and capabilities.

4.9 See also

- Process state

4.10 References

[1] “Preemptive multitasking”. *riscos.info*. 2009-11-03. Retrieved 2014-07-27.

[2] “UNIX, Part One”. *The Digital Research Initiative*. ibiblio.org. 2002-01-30. Retrieved 2014-01-09.

[3] Smart Computing Article - Windows 2000 &16-Bit Applications

[4] Eduardo Ciliendo; Takechika Kunimasa (April 25, 2008). “Linux Performance and Tuning Guidelines” (PDF). *redbooks.ibm.com*. IBM. p. 4. Retrieved March 1, 2015.

[5] “Context Switch Definition”. *linfo.org*. May 28, 2006. Retrieved February 26, 2015.

[6] “What are threads (user/kernel)?". *tldp.org*. September 8, 1997. Retrieved February 26, 2015.

Chapter 5

Time-sharing

This article is about the computing term. For the type of property ownership, see Timeshare.

In computing, **time-sharing** is the sharing of a computing resource among many users by means of multiprogramming and multi-tasking. Its introduction in the 1960s, and emergence as the prominent model of computing in the 1970s, represented a major technological shift in the history of computing.

By allowing a large number of users to interact concurrently with a single computer, time-sharing dramatically lowered the cost of providing computing capability, made it possible for individuals and organizations to use a computer without owning one, and promoted the interactive use of computers and the development of new interactive applications.

5.1 History

5.1.1 Batch processing

Main article: Batch processing

The earliest computers were extremely expensive devices, and very slow in comparison to recent models. Machines were typically dedicated to a particular set of tasks and operated by control panels, the operator manually entering small programs via switches in order to load and run a series of programs. These programs might take hours, or even weeks, to run. As computers grew in speed, run times dropped, and soon the time taken to start up the next program became a concern. Batch processing methodologies evolved to decrease these “dead periods” by queuing up programs so that as soon as one program completed, the next would start.

To support a batch processing operation, a number of comparatively inexpensive card punch or paper tape writers were used by programmers to write their programs “offline”. When typing (or punching) was complete, the programs were submitted to the operations team, which scheduled them to be run. Important programs were started quickly; how long before less important programs were started was unpredictable. When the program run was finally completed, the output (generally printed) was returned to the programmer. The complete process might take days, during which time the programmer might never see the computer.

The alternative of allowing the user to operate the computer directly was generally far too expensive to consider. This was because users might have long periods of entering code while the computer remained idle. This situation limited interactive development to those organizations that could afford to waste computing cycles: large universities for the most part. Programmers at the universities decried the behaviors that batch processing imposed, to the point that Stanford students made a short film humorously critiquing it.[1] They experimented with new ways to interact directly with the computer, a field today known as human–computer interaction.

5.1.2 Time-sharing

Unix time-sharing at the University of Wisconsin, 1978

Time-sharing was developed out of the realization that while any single user would make inefficient use of a computer, a large group of users together would not. This was due to the pattern of interaction: Typically an individual user entered bursts of information followed by long pauses but a group of users working at the *same time* would mean that the pauses of one user would be filled by the activity of the others. Given an optimal group size, the overall process could be very efficient. Similarly, small slices of time spent waiting for disk, tape, or network input could be granted to other users.

Implementing a system able to take advantage of this would be difficult. Batch processing was really a methodological development on top of the earliest systems; computers still ran single programs for single users at any time, all that batch processing changed was the time delay between one program and the next. Developing a system that supported multiple users at the same time was a completely different concept; the “state” of each user and their programs would have to be kept in the machine, and then switched between quickly. This would take up computer cycles, and on the slow machines of the era this was a concern. However, as computers rapidly improved in speed, and especially in size of core memory in which users’ states were retained, the overhead of time-sharing continually decreased, relatively.

The concept was first described publicly in early 1957 by Bob Bemer as part of an article in *Automatic Control Magazine*. The first project to implement a time-sharing system was initiated by John McCarthy in late 1957, on a modified IBM 704, and later on an additionally modified IBM 7090 computer. Although he left to work on Project MAC and other projects, one of the results of the project, known as the *Compatible Time-Sharing System* or CTSS, was demonstrated in November 1961. CTSS has a good claim to be the first time-sharing system and remained in use until 1973. Another contender for the first demonstrated time-sharing system was PLATO II, created by Donald Bitzer at a public demonstration at Robert Allerton Park near the University of Illinois in early 1961. Bitzer has long said that the PLATO project would have gotten the patent on time-sharing if only the University of Illinois had known how to process patent applications faster, but at the time university patents were so few and far between, they took a long time to be submitted. The first commercially successful time-sharing system was the Dartmouth Time Sharing System.

5.1.3 Development

Throughout the late 1960s and the 1970s, computer terminals were multiplexed onto large institutional mainframe computers (Centralized computing systems), which in many implementations sequentially polled the terminals to see if there was any additional data or action requested by the computer user. Later technology in interconnections were interrupt driven, and some of these used parallel data transfer technologies such as the IEEE 488 standard. Generally, computer terminals were utilized on college properties in much the same places as *desktop computers* or *personal computers* are found today. In the earliest days of personal computers, many were in fact used as particularly smart terminals for time-sharing systems.

With the rise of microcomputing in the early 1980s, time-sharing faded into the background because individual microprocessors were sufficiently inexpensive that a single person could have all the CPU time dedicated solely to their needs, even when idle. However, the Internet has brought the general concept of time-sharing back into popularity. Expensive corporate server farms costing millions can host thousands of customers all sharing the same common resources. As with the early serial terminals, websites operate primarily in bursts of activity followed by periods of idle time. This bursting nature permits the service to be used by many website customers at once, and none of them notice any delays in communications until the servers start to get very busy.

5.1.4 Time-sharing business

In the 1960s, several companies started providing time-sharing services as service bureaus. Early systems used Teletype Model 33 KSR or ASR or Teletype Model 35 KSR or ASR machines in ASCII environments, and IBM Selectric typewriter-based terminals in EBCDIC environments. They would connect to the central computer by dial-up Bell 103A modem or acoustically coupled modems operating at 10–15 characters per second. Later terminals and modems supported 30–120 characters per second. The time-sharing system would provide a complete operating environment, including a variety of programming language processors, various software packages, file storage, bulk printing, and off-line storage. Users were charged rent for the terminal, a charge for hours of connect time, a charge for seconds of CPU time, and a charge for kilobyte-months of disk storage.

Common systems used for time-sharing included the SDS 940, the PDP-10, and the IBM 360. Companies providing this service included GE's GEISCO, IBM subsidiary The Service Bureau Corporation, Tymshare (founded in 1966), National CSS (founded in 1967 and bought by Dun & Bradstreet in 1979), Dial Data (bought by Tymshare in 1968), and Bolt, Beranek, and Newman (BBN). By 1968, there were 32 such service bureaus serving the US National Institutes of Health (NIH) alone.[2] The *Auerbach Guide to Timesharing* (1973) lists 125 different timesharing services using equipment from Burroughs, CDC, DEC, HP, Honeywell, IBM, RCA, Univac and XDS.[3]

5.1.5 The computer utility

Beginning in 1964 the Multics operating system was designed as a computing utility, modeled on the electrical or telephone utilities. In the 1970s Ted Nelson's original "Xanadu" hypertext repository was envisioned as such a service. It seemed as the computer industry grew that no such consolidation of computing resources would occur as timesharing systems. However in the 1990s the concept was revived in somewhat modified form as cloud computing.

5.1.6 Security

Time-sharing was the first time that multiple processes, owned by different users, were running on a single machine; and these processes could interfere with one another.[4] For example, one process might alter shared resources which another process relied on, such as a variable stored in memory. When only one user was using the system, this would result in possibly wrong output - but with multiple users, this might mean that other users got to see information they were not meant to see.

To prevent this from happening, an operating system needed to enforce a set of policies that determined which privileges each process had. For example, the operating system might deny access to a certain variable by a certain process.

The first international conference on computer security in London in 1971 was primarily driven by the time-sharing industry and its customers.

5.2 Notable time-sharing systems

See also: Time-sharing system evolution

Significant early timesharing systems:[3]

- Allen-Babcock RUSH (Remote Users of Shared Hardware) Time-sharing System on IBM S/360 hardware[5] → Tymshare
- AT&T Bell Labs Unix → UC Berkeley BSD Unix
- BBN PDP-1 Time-sharing System → Massachusetts General Hospital PDP-1D → MUMPS
- BBN TENEX → DEC TOPS-20, Foonly FOONEX, MAXC OS at PARC, Stanford Low Overhead Timesharing System (LOTS)
- Berkeley Timesharing System at UC Berkeley Project Genie → Scientific Data Systems SDS 940 (Tymshare, BBN, SRI, Community Memory) → BCC 500 → MAXC at PARC
- Burroughs Time-sharing MCP → HP 3000 MPE
- Cambridge Multiple Access System was developed for the Titan, the prototype Atlas 2 computer built by Ferranti for the University of Cambridge.[6] This was the first time-sharing system developed outside the United States, and which influenced the later development of UNIX.
- CDC MACE, APEX → Kronos → NOS → NOS/VE
- CompuServe, also branded as Compu-Serv, CIS.
- Compu-Time, Inc.,[3] on Honeywell 400/4000, started in 1968 in Ft Lauderdale, Florida, moved to Daytona Beach in 1970.
- Dartmouth Time Sharing System (DTSS) → GE Time-sharing → GEnie
- DEC PDP-6 Time-sharing Monitor → TOPS-10 → TSS-8, RSTS-11, RSX-11 → VAX/VMS
- HP 2000 Time-Shared BASIC
- IBM CALL/360, CALL/OS - using IBM 360/50
- IBM CP-40 → CP-67 → CP-370 → CP/CMS → VM/CMS
- IBM TSO for OS/MVT → for OS/VS2 → for MVS → for z/OS
- IBM TSS/360 → TSS/370
- International Timesharing Corporation on dual CDC 3300 systems.[3]
- MIT CTSS → MULTICS (MIT / GE / Bell Labs) → Unix
- MIT Time-sharing System for the DEC PDP-1 → ITS
- McGill University MUSIC → IBM MUSIC/SP
- Michigan Terminal System, on the IBM S/360-67, S/370, and successors.
- Michigan State University CDC SCOPE/HUSTLER System
- National CSS VP/CSS, on IBM 360 series; originally based on IBM's CP/CMS.
- Oregon State University OS-3, on CDC 3000 series.
- Prime Computer PRIMOS
- RAND JOSS → JOSS-2 → JOSS-3
- RCA TSOS → Univac / Unisys VMOS → VS/9
- Service in Informatics and Analysis (SIA), on CDC 6600 Kronos.
- System Development Corporation Time-sharing System, on the AN/FSQ-32.
- Stanford ORVYL and WYLBUR, on IBM S/360-67.
- Stanford PDP-1 Time-sharing System → SAIL → WAITS
- Time Sharing Ltd. (TSL)[7] on DEC PDP-10 systems → Automatic Data Processing (ADP), first commercial time-sharing system in Europe and first dual (fault tolerant) time-sharing system.

- Tymshare SDS-940 → Tymcom X → Tymcom XX
- UC Berkeley CAL-TSS, on CDC 6400.
- XDS UTS → CP-V → Honeywell CP-6

5.3 See also

- The Heralds of Resource Sharing, a 1972 film.
- History of CP/CMS, IBM's virtual machine operating system (CP) that supported time-sharing (CMS).
- IBM M44/44X, an experimental computer system based on an IBM 7044 used to simulate multiple virtual machines.
- IBM System/360 Model 67, the only IBM S/360 series mainframe to support virtual memory.
- Multiseat configuration, multiple users on a single personal computer.
- Project MAC, a DARPA funded project at MIT famous for groundbreaking research in operating systems, artificial intelligence, and the theory of computation.
- TELCOMP, an interactive, conversational programming language based on JOSS, developed by BBN in 1964.
- Timeline of operating systems
- VAX (Virtual Address eXtension), a computer architecture and family of computers developed by DEC.
- Virtual memory

5.4 References

[1] Ellis D. Kropotchev Silent Film, 1967, This student-produced film from Stanford University is a humorous spoof of the trials and tribulations of a college hacker condemned to use batch processing. Originally created by Arthur Eisenson and Gary Feldman, the film gives the viewer a feel for the process of computer programming in the 1960s. Original music by Heather Yager. Computer History Museum, Object ID 102695643. Retrieved November 29, 2013.

[2] "Information Technology Corporate Histories Collection", Computer History Museum. Retrieved November 29, 2013.

[3] *Auerbach Guide to Time Sharing*. Auerbach Publishers, Inc. 1973. Retrieved November 29, 2013.

[4] Silberschatz, Abraham; Galvin, Peter; Gagne, Greg (2010). *Operating system concepts* (8th ed.). Hoboken, N.J.: Wiley & Sons. p. 591. ISBN 978-0-470-23399-3.

[5] "A Brief Description of Privacy Measures in the RUSH Time-Sharing System", J.D. Babcock, AFIPS Conference Proceedings, Spring Joint Computer Conference, Vol. 30, 1967, pp. 301-302.

[6] Hartley, D. F. (1968), *The Cambridge multiple-access system: user's reference manual*, Cambridge: Cambridge Univ. Press, ISBN 978-0901224002

[7] "Time Sharing", James Miller. Retrieved 30 November 2013.

5.5 Further reading

- Nelson, Theodor (1974). *Computer Lib: You Can and Must Understand Computers Now*; *Dream Machines*: "New Freedoms Through Computer Screens— A Minority Report". Self-published. ISBN 0-89347-002-3. pp. 56–57.

5.6 External links

- "Time Sharing Supervisor Programs", notes comparing the supervisor programs of CP-67, TSS/360, the Michigan Terminal System (MTS), and Multics by Michael T. Alexander, *Advanced Topics in Systems Programming* (1970, revised 1971), University of Michigan Engineering Summer Conference.
- "The Computer Utility As A Marketplace For Computer Services", Robert Frankston's MIT Master's Thesis, 1973.
- Reminiscences on the Theory of Time-Sharing by John McCarthy, 1983.
- Origins of timesharing by Bob Bemer.
- "40 years of Multics, 1969-2009", an interview with Professor Fernando J. Corbató on the history of Multics and origins of time-sharing, 2009.
- "Mainframe Computers: The Virtues of Sharing", Revolution: The First 2000 Years of Computing, Computer History Museum Exhibition, January 2011.
- "Mainframe Computers: Timesharing as a Business", Revolution: The First 2000 Years of Computing, Computer History Museum Exhibition, January 2011.

Chapter 6

Real-time computing

In computer science, **real-time computing** (**RTC**), or **reactive computing**, is the study of hardware and software systems that are subject to a "real-time constraint", for example operational deadlines from event to system response. Real-time programs must guarantee response within strict time constraints, often referred to as "deadlines".[1] Real-time responses are often understood to be in the order of milliseconds, and sometimes microseconds. Conversely, a system without real-time facilities, cannot *guarantee* a response within any timeframe (regardless of *actual* or *expected* response times).

A real-time system is one which "controls an environment by receiving data, processing them, and returning the results sufficiently quickly to affect the environment at that time."[2] This usage of "real-time" should not be confused with the two other legitimate uses of the term: in simulation the term means that the simulation's clock runs as fast as a real clock, while in processing and enterprise systems the term is used to mean "without perceivable delay".

Real-time software may use one or more of the following: synchronous programming languages, real-time operating systems, and real-time networks, each of which provide essential frameworks on which to build a real-time software application.

A real-time system may be one where its application can be considered (within context) to be mission critical. The anti-lock brakes on a car are a simple example of a real-time computing system: the real-time constraint in this system is the time in which the brakes must be released to prevent the wheel from locking.[3] Real-time computations can be said to have *failed* if they are not completed before their deadline, where their deadline is relative to an event. A real-time deadline must be met, regardless of system load.

6.1 History

The term *real-time* derives from its use in early simulation, in which a real-world process is simulated at a rate that matched that of the real process (now called real-time simulation to avoid ambiguity). Analog computers, most often, were capable of simulating at a much faster pace than real-time, a situation that could be just as dangerous as a slow simulation if it were not also recognized and accounted for.

Once when the MOS Technology 6502 (used in the Commodore 64 and Apple II), and later when the Motorola 68000 (used in the Macintosh, Atari ST, and Commodore Amiga) were popular, anybody could use their home computer as a real-time system. The possibility to deactivate other interrupts allowed for hard-coded loops with defined timing, and the low interrupt latency allowed the implementation of a real-time operating system, giving the user interface and the disk drives lower priority than the real-time thread. Compared to these the Programmable Interrupt Controller of the Intel CPUs (8086..80586) generates a very large latency and the Windows operating system is neither a real-time operating system nor does it allow a program to take over the CPU completely and use its own scheduler, without using native machine language and thus surpassing all interrupting Windows code. However, several coding libraries exist which offer real time capabilities in a high level language on a variety of operating systems, for example Java Real Time. The Motorola 68000 and subsequent family members (68010, 68020 etc.) also became popular with manufacturers of industrial control systems thanks to this facility. This application area is one in which real-time control offers genuine advantages in terms of process performance and safety.

6.2 Criteria for real-time computing

A system is said to be *real-time* if the total correctness of an operation depends not only upon its logical correctness, but also upon the time in which it is performed.[4] Real-time systems, as well as their deadlines, are classified by the consequence of missing a deadline:

- *Hard* – missing a deadline is a total system failure.
- *Firm* – infrequent deadline misses are tolerable, but may degrade the system's quality of service. The

usefulness of a result is zero after its deadline.

- *Soft* – the usefulness of a result degrades after its deadline, thereby degrading the system's quality of service.

Thus, the goal of a *hard real-time system* is to ensure that all deadlines are met, but for *soft real-time systems* the goal becomes meeting a certain subset of deadlines in order to optimize some application-specific criteria. The particular criteria optimized depend on the application, but some typical examples include maximizing the number of deadlines met, minimizing the lateness of tasks and maximizing the number of high priority tasks meeting their deadlines.

Hard real-time systems are used when it is imperative that an event be reacted to within a strict deadline. Such strong guarantees are required of systems for which not reacting in a certain interval of time would cause great loss in some manner, especially damaging the surroundings physically or threatening human lives (although the strict definition is simply that missing the deadline constitutes failure of the system). For example, a car engine control system is a hard real-time system because a delayed signal may cause engine failure or damage. Other examples of hard real-time embedded systems include medical systems such as heart pacemakers and industrial process controllers. Hard real-time systems are typically found interacting at a low level with physical hardware, in embedded systems. Early video game systems such as the Atari 2600 and Cinematronics vector graphics had hard real-time requirements because of the nature of the graphics and timing hardware.

In the context of multitasking systems the scheduling policy is normally priority driven (pre-emptive schedulers). Other scheduling algorithms include Earliest Deadline First, which, ignoring the overhead of context switching, is sufficient for system loads of less than 100%.[5] New overlay scheduling systems, such as an Adaptive Partition Scheduler assist in managing large systems with a mixture of hard real-time and non real-time applications.

Soft real-time systems are typically used to solve issues of concurrent access and the need to keep a number of connected systems up-to-date through changing situations. An example can be software that maintains and updates the flight plans for commercial airliners: the flight plans must be kept reasonably current, but they can operate with the latency of a few seconds. Live audio-video systems are also usually soft real-time; violation of constraints results in degraded quality, but the system can continue to operate and also recover in the future using workload prediction and reconfiguration methodologies.[6]

6.2.1 Real-time in digital signal processing

In a real-time digital signal processing (DSP) process, the analyzed (input) and generated (output) samples can be processed (or generated) continuously in the time it takes to input and output the same set of samples *independent* of the processing delay.[7] It means that the processing delay must be bounded even if the processing continues for an unlimited time. That means that the mean processing time per sample is no greater than the sampling period, which is the reciprocal of the sampling rate. This is the criterion whether the samples are grouped together in large segments and processed as blocks or are processed individually and whether there are long, short, or non-existent input and output buffers.

Consider an audio DSP example; if a process requires 2.01 seconds to analyze, synthesize, or process 2.00 seconds of sound, it is not real-time. If it takes 1.99 seconds, it is or can be made into a real-time DSP process.

A common life analog is standing in a line or queue waiting for the checkout in a grocery store. If the line asymptotically grows longer and longer without bound, the checkout process is not real-time. If the length of the line is bounded, customers are being "processed" and output as rapidly, on average, as they are being inputted and that process *is* real-time. The grocer might go out of business or must at least lose business if they cannot make their checkout process real-time; thus, it is fundamentally important that this process is real-time.

A signal processing algorithm that cannot keep up with the flow of input data with output falling farther and farther behind the input is not real-time. But if the delay of the output (relative to the input) is bounded regarding a process that operates over an unlimited time, then that signal processing algorithm is real-time, even if the throughput delay may be very long.

6.3 Real-time and high-performance

Real-time computing is sometimes misunderstood to be high-performance computing, but this is not an accurate classification.[8] For example, a massive supercomputer executing a scientific simulation may offer impressive performance, yet it is not executing a real-time computation. Conversely, once the hardware and software for an anti-lock braking system have been designed to meet its required deadlines, no further performance gains are obligatory. Furthermore, if a network server is highly loaded with network traffic, its response time may be slower but will (in most cases) still succeed before it times out (hits its deadline). Hence, such a network server would not be considered a real-time system: temporal failures (delays, time-outs, etc.) are typically small and compartmentalized (limited in effect) but are not catastrophic failures. In a real-time system, such as the FTSE 100 Index, a slow-down beyond limits would often be considered catastrophic in its application context. Therefore, the most important requirement of a real-time system is predictability and not performance.

Some kinds of software, such as many chess-playing programs, can fall into either category. For instance, a chess program designed to play in a tournament with a clock will need to decide on a move before a certain deadline or lose the game, and is therefore a real-time computation, but a chess program that is allowed to run indefinitely before moving is not. In both of these cases, however, high performance is desirable: the more work a tournament chess program can do in the allotted time, the better its moves will be, and the faster an unconstrained chess program runs, the sooner it will be able to move. This example also illustrates the essential difference between real-time computations and other computations: if the tournament chess program does not make a decision about its next move in its allotted time it loses the game—i.e., it fails as a real-time computation—while in the other scenario, meeting the deadline is assumed not to be necessary. High-performance is indicative of the amount of processing that is performed in a given amount of time, while real-time is the ability to get done with the processing to yield a useful output in the available time.

6.4 Near real-time

The term "*near real-time*" or "*nearly real-time*" (NRT), in telecommunications and computing, refers to the time delay introduced, by automated data processing or network transmission, between the occurrence of an event and the use of the processed data, such as for display or feedback and control purposes. For example, a near-real-time display depicts an event or situation as it existed at the current time minus the processing time, as nearly the time of the live event.[9]

The distinction between the terms “near real time” and “real time” is somewhat nebulous and must be defined for the situation at hand. The term implies that there are no significant delays.[9] In many cases, processing described as “real-time” would be more accurately described as “near real-time”.

Near real-time also refers to delayed real-time transmission of voice and video. It allows playing video images, in approximately real-time, without having to wait for an entire large video file to download. Incompatible databases can export/import to common flat files that the other database can import/export on a scheduled basis so that they can sync/share common data in “near real-time” with each other.

The distinction between “near real-time” and “real-time” varies, and the delay is dependent on the type and speed of the transmission. The delay in near real-time is typically of the order of several seconds to several minutes.

6.5 Design methods

Several methods exist to aid the design of real-time systems, an example of which is MASCOT, an old but very successful method which represents the concurrent structure of the system. Other examples are HOOD, Real-Time UML, AADL, the Ravenscar profile, and Real-Time Java.

6.6 See also

- Processing modes
- Real-time testing
- Synchronous programming language
- Ptolemy Project
- DSOS
- Worst-case execution time
- Scheduling analysis real-time systems
- Time-utility function
- Real-time computer graphics

6.7 References

[1] Ben-Ari, M., “Principles of Concurrent and Distributed Programming”, Prentice Hall, 1990. ISBN 0-13-711821-X. Ch16, Page 164

[2] Martin, James (1965). *Programming Real-time Computer Systems*. Englewood Cliffs, NJ: Prentice-Hall Inc. p. 4. ISBN 013-730507-9.

[3] Krishna Kant (May 2010). *Computer-Based Industrial Control*. *books.google.com* (PHI Learning). p. 356. Retrieved 2015-01-17.

[4] Shin, K.G.; Ramanathan, P. (Jan 1994). “Real-time computing: a new discipline of computer science and engineering”. *Proceedings of the IEEE* (IEEE) **82** (1): 6–24. doi:10.1109/5.259423. ISSN 0018-9219.

[5] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment. Journal of the ACM, 20(1):46-–61, Jan. 1973. http://citeseer.ist.psu.edu/liu73scheduling.html

[6] “Real-time reconfiguration for guaranteeing QoS provisioning levels in Grid environments”. *Future Generation Computer Systems* (Elsevier) **25** (7): 779–784. July 2009. doi:10.1016/j.future.2008.11.001.

[7] S.M. Kuo, B.H. Lee, and W. Tian, “Real-Time Digital Signal Processing: Implementations and Applications”, Wiley, 2006. ISBN 0-470-01495-4. Section 1.3.4: *Real-Time Constraints*.

[8] John Stankovic (1988), “Misconceptions about real-time computing: a serious problem for next-generation systems”, *Computer* (IEEE Computer Society) **21** (10): 11, doi:10.1109/2.7053

[9] “Federal Standard 1037C: Glossary of Telecommunications Terms”. Its.bldrdoc.gov. Retrieved 2014-04-26.

6.8 Further reading

- Alan Burns and Andy Wellings (2009), *Real-Time Systems and Programming Languages* (4th ed.), Addison-Wesley, ISBN 978-0-321-41745-9
- Buttazzo, Giorgio (2011), *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, New York, NY: Springer.
- Liu, Jane W.S. (2000), *Real-time systems*, Upper Saddle River, NJ: Prentice Hall.

6.9 External links

6.9.1 Technical committees

- IEEE Technical Committee on Real-Time Systems
- Euromicro Technical Committee on Real-time Systems

6.9.2 Scientific conferences

- RTNS - International Conference on Real-Time Networks and Systems
- ECRTS - Euromicro Conference on Real-time Systems
- IEEE Real-time Systems Symposium
- IEEE Real-time Technology and Applications Symposium
- International Symposium on Object-oriented Real-time distributed Computing
- IEEE International Conference on Embedded and Real-Time Computing Systems and Applications
- Real-Time & Embedded Computing Conference

6.9.3 Journals

- International Journal of Critical Computer-Based Systems
- International Journal of Real-Time Systems

6.9.4 Research groups

- Cyber-Physical Systems Integration Lab, University of Illinois at Urbana-Champaign.
- IDA Institute of Computer and Network Engineering, TU Braunschweig, Germany
- CISTER Research Unit, ISEP, Polytechnic Institute of Porto (IPP), Portugal
- Real-Time Systems Research Group,INRIA,LORIA NANCY, France
- Real-Time & Embedded Computing Laboratory (USMAN SHARIF BCS-SP03-37)
- Mälardalen Real-Time research Centre
- Real-Time Computing Laboratory
- Real-Time Systems Laboratory
- RTSE Laboratory
- Institute for Systems Engineering - Real Time systems Group
- Real-Time Systems Laboratory at Scuola Superiore Sant'Anna, Pisa, Italy
- Technical University of Kaiserslautern - Institute for Electrical Engineering and Information Technology - Real-Time Systems Group
- Vienna University of Technology - Institute for Computer Engineering - Real-Time Systems Group
- Real-Time Systems Research Group at the University of York, UK
- Chalmers University of Technology - Dependable Real-Time Systems research group
- ARTES: a national Swedish strategic research initiative in Real-Time Systems supported by the Swedish Foundation for Strategic Research (SSF), SE
- Real-Time Systems at the University of North Carolina at Chapel Hill
- Real-time Systems Laboratory at Virginia Polytechnic and State University, Blacksburg
- Mc2labs RealTime Industries. Real-time ICT Research & Development. RealTime Internet programming infrastructures, Mogliano Veneto, Italy

- Simula Research Laboratory, Media Performance Group
- Parallel Architectures for Real-time Systems, Brussels, Belgium, Europe.

6.9.5 Technical papers

- The What, Where and Why of Real-Time Simulation

Chapter 7

Fault tolerance

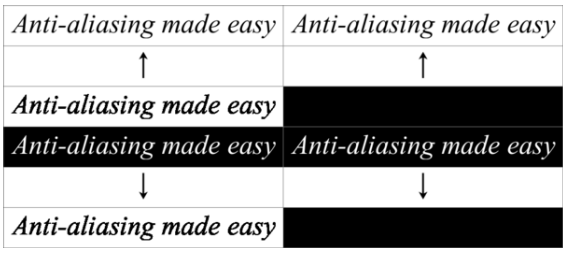
Fault tolerance is the property that enables a system to continue operating properly in the event of the failure of (or one or more faults within) some of its components. If its operating quality decreases at all, the decrease is proportional to the severity of the failure, as compared to a naively designed system in which even a small failure can cause total breakdown. Fault tolerance is particularly sought after in high-availability or life-critical systems.

A **fault-tolerant design** enables a system to continue its intended operation, possibly at a reduced level, rather than failing completely, when some part of the system fails.[1] The term is most commonly used to describe computer systems designed to continue more or less fully operational with, perhaps, a reduction in throughput or an increase in response time in the event of some partial failure. That is, the system as a whole is not stopped due to problems either in the hardware or the software. An example in another field is a motor vehicle designed so it will continue to be drivable if one of the tires is punctured. A structure is able to retain its integrity in the presence of damage due to causes such as fatigue, corrosion, manufacturing flaws, or impact.

Within the scope of an *individual* system, fault tolerance can be achieved by anticipating exceptional conditions and building the system to cope with them, and, in general, aiming for self-stabilization so that the system converges towards an error-free state. However, if the consequences of a system failure are catastrophic, or the cost of making it sufficiently reliable is very high, a better solution may be to use some form of duplication. In any case, if the consequence of a system failure is so catastrophic, the system must be able to use reversion to fall back to a safe mode. This is similar to roll-back recovery but can be a human action if humans are present in the loop.

7.1 Terminology

A highly fault-tolerant system might continue at the same level of performance even though one or more components have failed. For example, a building with a backup electrical generator will provide the same voltage to wall outlets even if the grid power fails.



An example of graceful degradation by design in an image with transparency. The top two images are each the result of viewing the composite image in a viewer that recognises transparency. The bottom two images are the result in a viewer with no support for transparency. Because the transparency mask (centre bottom) is discarded, only the overlay (centre top) remains; the image on the left has been designed to degrade gracefully, hence is still meaningful without its transparency information.

A system that is designed to fail safe, or fail-secure, or **fail gracefully**, whether it functions at a reduced level or fails completely, does so in a way that protects people, property, or data from injury, damage, intrusion, or disclosure. In computers, a program might fail-safe by executing a graceful exit (as opposed to an uncontrolled crash) in order to prevent data corruption after experiencing an error. A similar distinction is made between “failing well” and "failing badly".

Fail-deadly is the opposite strategy, which can be used in weapon systems that are designed to kill or injure targets even if part of the system is damaged or destroyed.

A system that is designed to experience **graceful degradation**, or to **fail soft** (used in computing, similar to “fail safe”[2]) operates at a reduced level of performance after some component failures. For example, a building may operate lighting at reduced levels and elevators at reduced speeds if grid power fails, rather than either trapping people in the dark completely or continuing to operate at full power. In computing an example of graceful degradation is that if insufficient network bandwidth is available to stream an online video, a lower-resolution version might be streamed in place of the high-resolution version. Progressive enhancement is an example in computing, where web pages are available in a basic functional format for older, small-screen, or limited-capability web browsers, but in an enhanced version for browsers capable of handling additional technologies or that have a larger display available.

In fault-tolerant computer systems, programs that are considered robust are designed to continue operation despite an error, exception, or invalid input, instead of crashing completely. Software brittleness is the opposite of robustness. Resilient networks continue to transmit data despite the failure of some links or nodes; resilient buildings and infrastructure are likewise expected to prevent complete failure in situations like earthquakes, floods, or collisions.

A system with high failure transparency will alert users that a component failure has occurred, even if it continues to operate with full performance, so that failure can be repaired or imminent complete failure anticipated. Likewise, a fail-fast component is designed to report at the first point of failure, rather than allow downstream components to fail and generate reports then. This allows easier diagnosis of the underlying problem, and may prevent improper operation in a broken state.

7.2 Components

If each component, in turn, can continue to function when one of its subcomponents fails, this will allow the total system to continue to operate as well. Using a passenger vehicle as an example, a car can have "run-flat" tires, which each contain a solid rubber core, allowing them to be used even if a tire is punctured. The punctured "run-flat" tire may be used for a limited time at a reduced speed.

7.3 Redundancy

Main article: redundancy (engineering)

Redundancy is the provision of functional capabilities that would be unnecessary in a fault-free environment.[3] This can consist of backup components which automatically "kick in" should one component fail. For example, large cargo trucks can lose a tire without any major consequences. They have many tires, and no one tire is critical (with the exception of the front tires, which are used to steer). The idea of incorporating redundancy in order to improve the reliability of a system was pioneered by John von Neumann in the 1950s.[4]

Two kinds of redundancy are possible:[5] space redundancy and time redundancy. Space redundancy provides additional components, functions, or data items that are unnecessary for fault-free operation. Space redundancy is further classified into hardware, software and information redundancy, depending on the type of redundant resources added to the system. In time redundancy the computation or data transmission is repeated and the result is compared to a stored copy of the previous result.

7.4 Criteria

Providing fault-tolerant design for every component is normally not an option. Associated redundancy brings a number of penalties: increase in weight, size, power consumption, cost, as well as time to design, verify, and test. Therefore, a number of choices have to be examined to determine which components should be fault tolerant:[6]

- **How critical is the component?** In a car, the radio is not critical, so this component has less need for fault tolerance.

- **How likely is the component to fail?** Some components, like the drive shaft in a car, are not likely to fail, so no fault tolerance is needed.

- **How expensive is it to make the component fault tolerant?** Requiring a redundant car engine, for example, would likely be too expensive both economically and in terms of weight and space, to be considered.

An example of a component that passes all the tests is a car's occupant restraint system. While we do not normally think of the *primary* occupant restraint system, it is gravity. If the vehicle rolls over or undergoes severe g-forces, then this primary method of occupant restraint may fail. Restraining the occupants during such an accident is absolutely critical to safety, so we pass the first test. Accidents causing occupant ejection were quite common before seat belts, so we pass the second test. The cost of a redundant restraint method like seat belts is quite low, both economically and in terms or weight and space, so we pass the third test. Therefore, adding seat belts to all vehicles is an excellent idea. Other "supplemental restraint systems", such as airbags, are more expensive and so pass that test by a smaller margin.

7.5 Requirements

The basic characteristics of fault tolerance require:

1. No single point of failure – If a system experiences a failure, it must continue to operate without interruption during the repair process.

2. Fault isolation to the failing component – When a failure occurs, the system must be able to isolate the failure to the offending component. This requires

the addition of dedicated failure detection mechanisms that exist only for the purpose of fault isolation. Recovery from a fault condition requires classifying the fault or failing component. The National Institute of Standards and Technology (NIST) categorizes faults based on locality, cause, duration, and effect.

3. Fault containment to prevent propagation of the failure – Some failure mechanisms can cause a system to fail by propagating the failure to the rest of the system. An example of this kind of failure is the "rogue transmitter" which can swamp legitimate communication in a system and cause overall system failure. Firewalls or other mechanisms that isolate a rogue transmitter or failing component to protect the system are required.

4. Availability of reversion modes

In addition, fault-tolerant systems are characterized in terms of both planned service outages and unplanned service outages. These are usually measured at the application level and not just at a hardware level. The figure of merit is called availability and is expressed as a percentage. For example, a five nines system would statistically provide 99.999% availability.

Fault-tolerant systems are typically based on the concept of redundancy.

7.6 Replication

Spare components address the first fundamental characteristic of fault tolerance in three ways:

- Replication: Providing multiple identical instances of the same system or subsystem, directing tasks or requests to all of them in parallel, and choosing the correct result on the basis of a quorum;
- Redundancy: Providing multiple identical instances of the same system and switching to one of the remaining instances in case of a failure (failover);
- Diversity: Providing multiple *different* implementations of the same specification, and using them like replicated systems to cope with errors in a specific implementation.

All implementations of RAID, redundant array of independent disks, except RAID 0, are examples of a fault-tolerant storage device that uses data redundancy.

A lockstep fault-tolerant machine uses replicated elements operating in parallel. At any time, all the replications of each element should be in the same state. The same inputs are provided to each replication, and the same outputs are expected. The outputs of the replications are compared using a voting circuit. A machine with two replications of each element is termed dual modular redundant (DMR). The voting circuit can then only detect a mismatch and recovery relies on other methods. A machine with three replications of each element is termed triple modular redundant (TMR). The voting circuit can determine which replication is in error when a two-to-one vote is observed. In this case, the voting circuit can output the correct result, and discard the erroneous version. After this, the internal state of the erroneous replication is assumed to be different from that of the other two, and the voting circuit can switch to a DMR mode. This model can be applied to any larger number of replications.

Lockstep fault-tolerant machines are most easily made fully synchronous, with each gate of each replication making the same state transition on the same edge of the clock, and the clocks to the replications being exactly in phase. However, it is possible to build lockstep systems without this requirement.

Bringing the replications into synchrony requires making their internal stored states the same. They can be started from a fixed initial state, such as the reset state. Alternatively, the internal state of one replica can be copied to another replica.

One variant of DMR is **pair-and-spare**. Two replicated elements operate in lockstep as a pair, with a voting circuit that detects any mismatch between their operations and outputs a signal indicating that there is an error. Another pair operates exactly the same way. A final circuit selects the output of the pair that does not proclaim that it is in error. Pair-and-spare requires four replicas rather than the three of TMR, but has been used commercially.

7.7 Disadvantages

Fault-tolerant design's advantages are obvious, while many of its disadvantages are not:

- **Interference with fault detection in the same component.** To continue the above passenger vehicle example, with either of the fault-tolerant systems it may not be obvious to the driver when a tire has been punctured. This is usually handled with a separate "automated fault-detection system". In the case of the tire, an air pressure monitor detects the loss of pressure and notifies the driver. The alternative is a "manual fault-detection system", such as manually inspecting all tires at each stop.

- **Interference with fault detection in another component.** Another variation of this problem is when fault tolerance in one component prevents fault detection in a different component. For example, if component B performs some operation based on the output from component A, then fault tolerance in B can hide a problem with A. If component B is later changed (to a less fault-tolerant design) the system may fail suddenly, making it appear that the new component B is the problem. Only after the system has been carefully scrutinized will it become clear that the root problem is actually with component A.

- **Reduction of priority of fault correction.** Even if the operator is aware of the fault, having a fault-tolerant system is likely to reduce the importance of repairing the fault. If the faults are not corrected, this will eventually lead to system failure, when the fault-tolerant component fails completely or when all redundant components have also failed.
- **Test difficulty.** For certain critical fault-tolerant systems, such as a nuclear reactor, there is no easy way to verify that the backup components are functional. The most infamous example of this is Chernobyl, where operators tested the emergency backup cooling by disabling primary and secondary cooling. The backup failed, resulting in a core meltdown and massive release of radiation.
- **Cost.** Both fault-tolerant components and redundant components tend to increase cost. This can be a purely economic cost or can include other measures, such as weight. Manned spaceships, for example, have so many redundant and fault-tolerant components that their weight is increased dramatically over unmanned systems, which don't require the same level of safety.
- **Inferior components.** A fault-tolerant design may allow for the use of inferior components, which would have otherwise made the system inoperable. While this practice has the potential to mitigate the cost increase, use of multiple inferior components may lower the reliability of the system to a level equal to, or even worse than, a comparable non-fault-tolerant system.

7.8 Examples

Hardware fault tolerance sometimes requires that broken parts be taken out and replaced with new parts while the system is still operational (in computing known as *hot swapping*). Such a system implemented with a single backup is known as **single point tolerant**, and represents the vast majority of fault-tolerant systems. In such systems the mean time between failures should be long enough for the operators to have time to fix the broken devices (mean time to repair) before the backup also fails. It helps if the time between failures is as long as possible, but this is not specifically required in a fault-tolerant system.

Fault tolerance is notably successful in computer applications. Tandem Computers built their entire business on such machines, which used single-point tolerance to create their **NonStop** systems with uptimes measured in years.

Fail-safe architectures may encompass also the computer software, for example by process replication (computer science).

Data formats may also be designed to degrade gracefully. HTML for example, is designed to be forward compatible, allowing new HTML entities to be ignored by Web browsers which do not understand them without causing the document to be unusable.

7.9 Related terms

There is a difference between fault tolerance and systems that rarely have problems. For instance, the Western Electric crossbar systems had failure rates of two hours per forty years, and therefore were highly *fault resistant*. But when a fault did occur they still stopped operating completely, and therefore were not *fault tolerant*.

7.10 See also

- Control reconfiguration
- Defence in depth
- Elegant degradation
- Error-tolerant design (human-error-tolerant design)
- Fault-tolerant computer systems
- List of system quality attributes
- Resilience (ecology)
- Resilience (network)
- Safe-life design

7.11 References

[1] Johnson, B. W. (1984). “Fault-Tolerant Microprocessor-Based Systems”, IEEE Micro, vol. 4, no. 6, pp. 6–21

[2] Stallings, W (2009): Operating Systems. Internals and Design Principles, *sixth edition*

[3] Laprie, J. C. (1985). “Dependable Computing and Fault Tolerance: Concepts and Terminology”, Proceedings of 15th International Symposium on Fault-Tolerant Computing (FTSC-15), pp. 2–11

[4] von Neumann, J. (1956). “Probabilistic Logics and Synthesis of Reliable Organisms from Unreliable Components”, in Automata Studies, eds. C. Shannon and J. McCarthy, Princeton University Press, pp. 43–98

[5] Avizienis, A. (1976). “Fault-Tolerant Systems”, IEEE Transactions on Computers, vol. 25, no. 12, pp. 1304–1312

[6] Dubrova, E. (2013). “Fault-Tolerant Design”, Springer, 2013, ISBN 978-1-4614-2112-2

7.12 Bibliography

- Brian Randell, P.A. Lee, P. C. Treleaven (June 1978). “Reliability Issues in Computing System Design”. *ACM Computing Surveys (CSUR)* **10** (2): 123–165. doi:10.1145/356725.356729. ISSN 0360-0300.

- P. J. Denning (December 1976). “Fault tolerant operating systems”. *ACM Computing Surveys (CSUR)* **8** (4): 359–389. doi:10.1145/356678.356680. ISSN 0360-0300.

- Theodore A. Linden (December 1976). “Operating System Structures to Support Security and Reliable Software”. *ACM Computing Surveys (CSUR)* **8** (4): 409–445. doi:10.1145/356678.356682. ISSN 0360-0300.

- Menychtas, Andreas; Konstanteli, Kleopatra (2012), *Fault Detection and Recovery Mechanisms and Techniques for Service Oriented Infrastructures*, Achieving Real-Time in Distributed Computing: From Grids to Clouds, IGI Global, pp. 259–274, doi:10.4018/978-1-60960-827-9.ch014

7.13 External links

- Implementation and evaluation of failsafe computer-controlled systems
- Seminar on Self-Healing Systems
- Interview with Robert Hanmer about his book *Patterns for Fault Tolerant Software* (Part One, Part Two) (Podcast)
- Article "Practical Considerations in Making CORBA Services Fault-Tolerant" by Priya Narasimhan
- Article "Experiences, Strategies and Challenges in Building Fault-Tolerant CORBA Systems" by Pascal Felber and Priya Narasimhan
- Dependability And Its Threats: A Taxonomy by Algirdas Avizienis, Jean-Claude Laprie, B. Randell
- EU funded research project HPC4U addressing development of fault tolerant technologies for Grid computing environments
- Fault Tolerance and High Availability Systems
- Graceful Degradation in the RKBExplorer
- Fault Tolerance and High Availability Systems for Check Point Firewall and VPN networks with Resilience line of FCR appliances
- Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Chapter 8

Mean time between failures

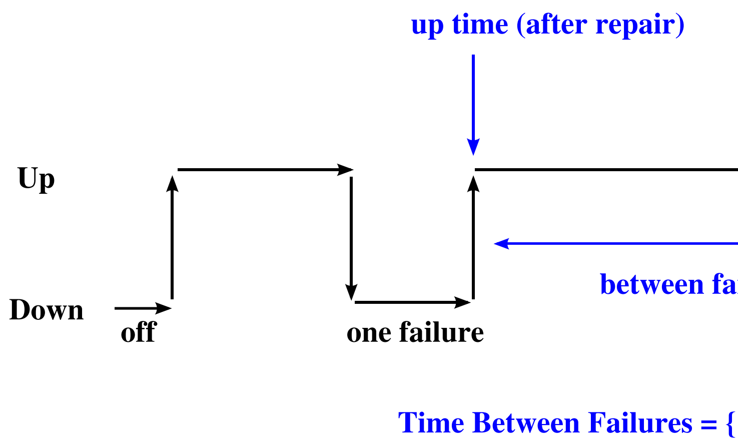
Mean time between failures (MTBF) is the predicted elapsed time between inherent failures of a system during operation.[1] MTBF can be calculated as the arithmetic mean (average) time between failures of a system. The MTBF is typically part of a model that assumes the failed system is immediately repaired (mean time to repair, or MTTR), as a part of a renewal process. This is in contrast to the mean time to failure (MTTF), which measures average time to failures with the modeling assumption that the failed system is not repaired (infinite repair time).

The definition of MTBF depends on the definition of what is considered a system failure. For complex, repairable systems, failures are considered to be those out of design conditions which place the system out of service and into a state for repair. Failures which occur that can be left or maintained in an unrepaired condition, and do not place the system out of service, are not considered failures under this definition.[2] In addition, units that are taken down for routine scheduled maintenance or inventory control are not considered within the definition of failure.

8.1 The Overview

Mean time between failures (MTBF) describes the expected time between two failures for a repairable system, while mean time to failure (MTTF) denotes the expected time to failure for a non-repairable system. For example, three identical systems starting to function properly at time 0 are working until all of them fail. The first system failed at 100 hours, the second failed at 120 hours and the third failed at 130 hours. The MTBF of the system is the average of the three failure times, which is 116.667 hours. If the systems are non-repairable, then their MTTF would be 116.667 hours.

In general, MTBF is the "up-time" between two failure states of a repairable system during operation as outlined here:



For each observation, the "down time" is the instantaneous time it went down, which is after (i.e. greater than) the moment it went up, the "up time". The difference ("down time" minus "up time") is the amount of time it was operating between these two events.

Once the MTBF of a system is known, the probability that any one particular system will be operational at time equal to the MTBF can be calculated. This calculation requires that the system is working within its "useful life period", which is characterized by a relatively constant failure rate (the middle part of the "bathtub curve") when only random failures are occurring. Under this assumption, any one particular system will survive to its calculated MTBF with a probability of 36.8% (i.e., it will fail before with a probability of 63.2%). The same applies to the MTTF of a system working within this time period.[3]

MTBF value prediction is an important element in the development of products. However, it is incorrect to extrapolate MTBF to give an estimate of the life time of a component, which will typically be much less than suggested by the original MTBF due to the much higher failure rates in the "end-of-life wearout" part of the "bathtub curve".

Reliability engineers and design engineers often use reliability software to calculate a product's MTBF according to various methods and standards (MIL-HDBK-217F, Telcordia SR332, Siemens Norm, FIDES,UTE 80-810

(RDF2000), etc.). However, these "prediction" methods are not intended to reflect fielded MTBF as is commonly believed; the intent of these tools is to focus design efforts on the weak links in the design.

8.2 Formal definition of MTBF

By referring to the figure above, the MTBF is the sum of the operational periods divided by the number of observed failures. If the "Down time" (with space) refers to the start of "downtime" (without space) and "up time" (with space) refers to the start of "uptime" (without space), the formula will be:

$$\text{failures between time Mean} = \text{MTBF} = \frac{\sum(\text{downtime of start} - \text{uptime of start})}{\text{failures of number}}.$$

The MTBF is often denoted by the Greek letter θ, or

$$\text{MTBF} = \theta.$$

The MTBF can be defined in terms of the expected value of the density function $f(t)$

$$\text{MTBF} = \int_0^\infty t f(t)\,dt$$

where f is the density function of time until failure – satisfying the standard requirement of density functions –

$$\int_0^\infty f(t)\,dt = 1.$$

In this context (of reliability) is density function $f(t)$ also often referred as reliability function $R(t)$.

8.3 Variations of MTBF

There are many variations of MTBF, such as *mean time between system aborts* (MTBSA) or *mean time between critical failures* (MTBCF) or *mean time between unscheduled removal* (MTBUR). Such nomenclature is used when it is desirable to differentiate among types of failures, such as critical and non-critical failures. For example, in an automobile, the failure of the FM radio does not prevent the primary operation of the vehicle. *Mean time to failure* (MTTF) is sometimes used instead of MTBF in cases where a system is replaced after a failure, since MTBF denotes time between failures in a system which is repaired. MTTFd is an extension of MTTF, where MTTFd is only concerned about failures which would result in a dangerous condition.

8.3.1 MTTF and MTTFd calculation

$$\text{MTTF} \approx \frac{B_{10}}{0.1 n_{op}},$$

$$\text{MTTFd} \approx \frac{B_{10d}}{0.1 n_{op}},$$

where B_{10} is the number of operations that a device will operate prior to 10% of a sample of those devices would fail. B_{10d} is the same calculation, but where 10% of the sample would fail to danger. n_{op} is the number of operations/cycles.[4]

8.4 Notes

[1] Jones, James V., *Integrated Logistics Support Handbook*, page 4.2

[2] Colombo, A.G., and Sáiz de Bustamante, Amalio: *Systems reliability assessment – Proceedings of the Ispra Course held at the Escuela Tecnica Superior de Ingenieros Navales, Madrid, Spain, September 19–23, 1988 in collaboration with Universidad Politecnica de Madrid*, 1988

[3] "Reliability and MTBF Overview". Vicor Reliability Engineering. Retrieved 1 November 2014.

[4] "B10d Assessment - Reliability Parameter for Electro-Mechanical Components". TUVRheinland. Retrieved 16 April 2012.

8.5 See also

- Failure rate
- Annualized failure rate
- Mean time to repair
- Power-On Hours

8.6 References

- Jones, James V., *Integrated Logistics Support Handbook*, McGraw–Hill Professional, 3rd edition (June 8, 2006), ISBN 0-07-147168-5

8.7 External links

- MTBF and Life Expectancy on Monitor
- MTBF and Life Expectancy on I/O Modules
- MTBF and Life Expectancy on Harddisk
- MTBF and Life Expectancy on Mechanical Craft
- Reliability and Availability Basics

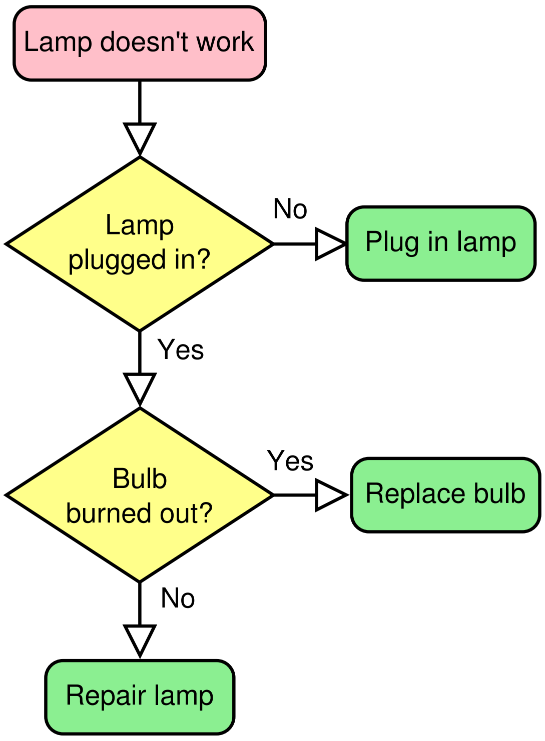
- Summary including MTTF discussion
- MTBCF Example spreadsheet
- “Failure Rates, MTBF, and All That” at MathPages

Chapter 9

Flowchart

For the music group, see Flowchart (band).

A **flowchart** is a type of diagram that represents an algorithm, workflow or process, showing the steps as boxes of various kinds, and their order by connecting them with arrows. This diagrammatic representation illustrates a solution model to a given problem. Flowcharts are used in analyzing, designing, documenting or managing a process or program in various fields.[1]



A simple flowchart representing a process for dealing with a non-functioning lamp.

9.1 Overview

Flowcharts are used in designing and documenting complex processes or programs. Like other types of diagrams, they help visualize what is going on and thereby help the people to understand a process, and perhaps also find flaws, bottlenecks, and other less-obvious features within it. There are many different types of flowcharts, and each type has its own repertoire of boxes and notational conventions. The two most common types of boxes in a flowchart are:

- a processing step, usually called *activity*, and denoted as a rectangular box
- a decision, usually denoted as a diamond.

A flowchart is described as “cross-functional” when the page is divided into different swimlanes describing the control of different organizational units. A symbol appearing in a particular “lane” is within the control of that organizational unit. This technique allows the author to locate the responsibility for performing an action or making a decision correctly, showing the responsibility of each organizational unit for different parts of a single process.

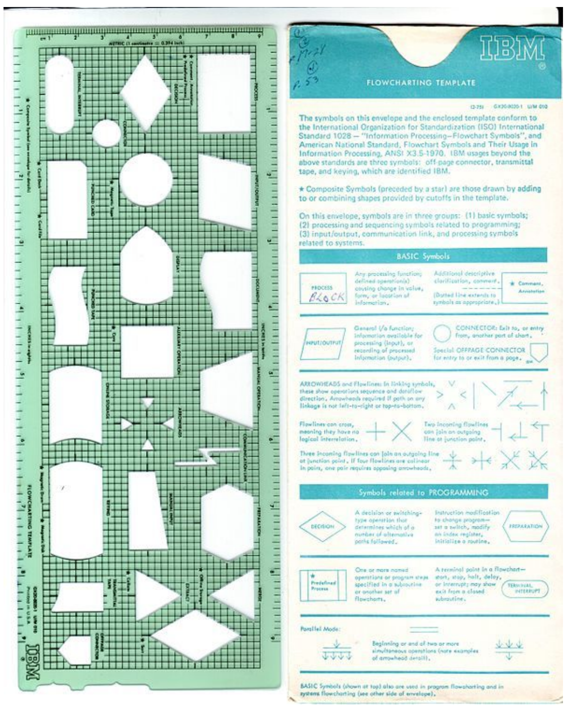
Flowcharts depict certain aspects of processes and they are usually complemented by other types of diagram. For instance, Kaoru Ishikawa defined the flowchart as one of the seven basic tools of quality control, next to the histogram, Pareto chart, check sheet, control chart, cause-and-effect diagram, and the scatter diagram. Similarly, in UML, a standard concept-modeling notation used in software development, the activity diagram, which is a type of flowchart, is just one of many different diagram types.

Nassi-Shneiderman diagrams and Drakon-charts are an alternative notation for process flow.

Common alternative names include: flowchart, process flowchart, functional flowchart, process map, process chart, functional process chart, business process model, process model, process flow diagram, work flow diagram, business flow diagram. The terms “flowchart” and “flow chart” are used interchangeably.

The underlying graph structure of a flow chart is a flow graph, which abstracts away node types, their contents and other ancillary information.

9.2 History



Template for drawing flowcharts (late 1970s) showing the different symbols.

The first structured method for document process flow, the "flow process chart", was introduced by Frank Gilbreth to members of the American Society of Mechanical Engineers (ASME) in 1921 in the presentation “Process Charts—First Steps in Finding the One Best Way”.[2] Gilbreth’s tools quickly found their way into industrial engineering curricula. In the early 1930s, an industrial engineer, Allan H. Mogensen began training business people in the use of some of the tools of industrial engineering at his Work Simplification Conferences in Lake Placid, New York.

A 1944 graduate of Mogensen's class, Art Spinanger, took the tools back to Procter and Gamble where he developed their Deliberate Methods Change Program. Another 1944 graduate, Ben S. Graham, Director of Formcraft Engineering at Standard Register Industrial, adapted the flow process chart to information processing with his development of the multi-flow process chart to display multiple documents and their relationships.[3] In 1947, ASME adopted a symbol set derived from Gilbreth’s original work as the “ASME Standard: Operation and Flow Process Charts.”[4]

Douglas Hartree in 1949 explained that Herman Goldstine and John von Neumann had developed a flowchart (originally, diagram) to plan computer programs.[5] His contemporary account is endorsed by IBM engineers[6] and by Goldstine’s personal recollections.[7] The original programming flowcharts of Goldstine and von Neumann can be seen in their unpublished report, “Planning and coding of problems for an electronic computing instrument, Part II, Volume 1” (1947), which is reproduced in von Neumann’s collected works.[8] Besides describing the logical flow of control, flowcharts allowed programmers to lay out machine language programs in computer memory before the development of assembly languages and assemblers.[9]

Flowcharts used to be a popular means for describing computer algorithms and are still used for this purpose.[10] Modern techniques such as UML activity diagrams and Drakon-charts can be considered to be extensions of the flowchart. In the 1970s the popularity of flowcharts as an own method decreased when interactive computer terminals and third-generation programming languages became the common tools of the trade, since algorithms can be expressed much more concisely as source code in such a language, and also because designing algorithms using flowcharts was more likely to result in spaghetti code because of the need for gotos to describe arbitrary jumps in control flow. Often pseudo-code is used, which uses the common idioms of such languages without strictly adhering to the details of a particular one.

9.3 Flowchart building blocks

9.3.1 Symbols

FlowChart Symbols List A typical flowchart from older basic computer science textbooks may have the following kinds of symbols:

Start and end symbols Represented as circles, ovals or rounded (fillet) rectangles, usually containing the word “Start” or “End”, or another phrase signaling the start or end of a process, such as “submit inquiry” or “receive product”.

Arrows Showing "flow of control". An arrow coming from one symbol and ending at another symbol represents that control passes to the symbol the arrow points to. The line for the arrow can be solid or dashed. The meaning of the arrow with dashed line may differ from one flowchart to another and can be defined in the legend.

Generic processing steps Represented as rectangles. Examples: “Add 1 to X"; “replace identified part"; “save changes” or similar.

Subroutines Represented as rectangles with double-struck vertical edges; these are used to show complex processing steps which may be detailed in a separate flowchart. Example: PROCESS-FILES. One subroutine may have multiple distinct entry points or exit flows (see coroutine); if so, these are shown as labeled 'wells' in the rectangle, and control arrows connect to these 'wells'.

Input/Output Represented as a parallelogram. Examples: Get X from the user; display X.

Prepare conditional Represented as a hexagon. Shows operations which have no effect other than preparing a value for a subsequent conditional or decision step (see below).

Conditional or decision Represented as a diamond (rhombus) showing where a decision is necessary, commonly a Yes/No question or True/False test. The conditional symbol is peculiar in that it has two arrows coming out of it, usually from the bottom point and right point, one corresponding to Yes or True, and one corresponding to No or False. (The arrows should always be labeled.) More than two arrows can be used, but this is normally a clear indicator that a complex decision is being taken, in which case it may need to be broken-down further or replaced with the “pre-defined process” symbol.

Junction symbol Generally represented with a black blob, showing where multiple control flows converge in a single exit flow. A junction symbol will have more than one arrow coming into it, but only one going out.

In simple cases, one may simply have an arrow point to another arrow instead. These are useful to represent an iterative process (what in Computer Science is called a loop). A loop may, for example, consist of a connector where control first enters, processing steps, a conditional with one arrow exiting the loop, and one going back to the connector.

For additional clarity, wherever two lines accidentally cross in the drawing, one of them may be drawn with a small semicircle over the other, showing that no junction is intended.

Labeled connectors Represented by an identifying label inside a circle. Labeled connectors are used in complex or multi-sheet diagrams to substitute for arrows. For each label, the “outflow” connector must always be unique, but there may be any number of “inflow” connectors. In this case, a junction in control flow is implied.

Concurrency symbol Represented by a double transverse line with any number of entry and exit arrows. These symbols are used whenever two or more control flows must operate simultaneously. The exit flows are activated concurrently when all of the entry flows have reached the concurrency symbol. A concurrency symbol with a single entry flow is a *fork*; one with a single exit flow is a *join*.

All processes should flow from top to bottom and left to right.

9.3.2 Data-flow extensions

A number of symbols have been standardized for data flow diagrams to represent data flow, rather than control flow. These symbols may also be used in control flowcharts (e.g. to substitute for the parallelogram symbol).

- A *Document* represented as a rectangle with a wavy base;
- A *Manual input* represented by quadrilateral, with the top irregularly sloping up from left to right. An example would be to signify data-entry from a form;
- A *Manual operation* represented by a trapezoid with the longest parallel side at the top, to represent an operation or adjustment to process that can only be made manually.
- A *Data File* represented by a cylinder.

9.4 Types of flowchart

Sterneckert (2003) suggested that flowcharts can be modeled from the perspective of different user groups (such as managers, system analysts and clerks) and that there are four general types:[11]

- *Document flowcharts*, showing controls over a document-flow through a system
- *Data flowcharts*, showing controls over a data-flow in a system
- *System flowcharts* showing controls at a physical or resource level
- *Program flowchart*, showing the controls in a program within a system

Notice that every type of flowchart focuses on some kind of control, rather than on the particular flow itself.[11]

However there are several of these classifications. For example Andrew Veronis (1978) named three basic types of flowcharts: the *system flowchart*, the *general flowchart*, and the *detailed flowchart*.[12] That same year Marilyn Bohl (1978) stated “in practice, two kinds of flowcharts are used in solution planning: *system flowcharts* and *program flowcharts*...”.[13] More recently Mark A. Fryman (2001) stated that there are more differences: “Decision flowcharts, logic flowcharts, systems flowcharts, product flowcharts, and process flowcharts are just a few of the different types of flowcharts that are used in business and government”.[14]

In addition, many diagram techniques exist that are similar to flowcharts but carry a different name, such as UML activity diagrams.

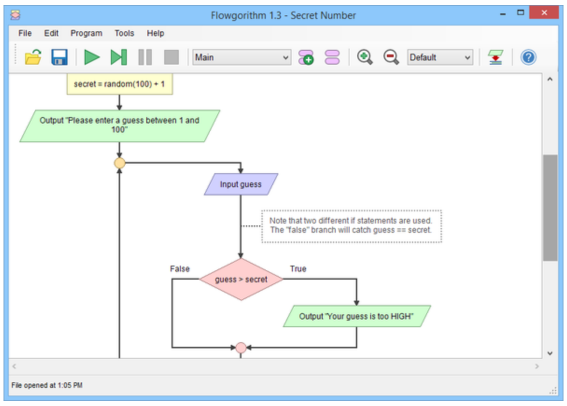
9.5 Software

9.5.1 Diagramming

Main article: Diagramming software § Flowchart

Any drawing program can be used to create flowchart diagrams, but these will have no underlying data model to share data with databases or other programs such as project management systems or spreadsheet. Some tools offer special support for flowchart drawing. Many software packages exist that can create flowcharts automatically, either directly from a programming language source code, or from a flowchart description language. On-line web-based versions of such programs are available.

9.5.2 Programming


Flowgorithm 1.3 - Secret Number
File
Edit
Program
Tools
Help
Main
Default
secret = random(100) + 1
Output "Please enter a guess between 1 and 100"
Input guess
False
guess > secret
True
Output "Your guess is too HIGH"

Note that two different if statements are used. The "false" branch will catch guess == secret.

File opened at 1:05 PM

Flowgorithm

There are several applications that use flowcharts to represent and execute programs. Generally these are used as teaching tools for beginner students.

These include:

- Flowgorithm
- Raptor
- LARP
- Visual Logic

9.6 See also

9.7 References

[1] SEVOCAB: Software and Systems Engineering Vocabulary. Term: *Flow chart*. Retrieved 31 July 2008.

[2] Frank Bunker Gilbreth, Lillian Moller Gilbreth (1921) *Process Charts*. *American Society of Mechanical Engineers.*

[3] Graham, Jr., Ben S. (10 June 1996). “People come first”. *Keynote Address at* Workflow Canada.

[4] American Society of Mechanical Engineers (1947) *ASME standard; operation and flow process charts*. New York, 1947. (online version)

[5] Hartree, Douglas (1949). *Calculating Instruments and Machines*. The University of Illinois Press. p. 112. Hartree stated:
“Von Neumann and Goldstine (40) have proposed a method of indicating the structure of the sequence of operating instructions by means of a “flow diagram” representing the control sequence. This is in the form of a block diagram, in which the blocks represent operations or groups of operations, and are joined by directed lines representing the sequence of these operations...”

[6] Bashe, Charles (1986). *IBM’s Early Computers*. The MIT Press. p. 327.

[7] Goldstine, Herman (1972). *The Computer from Pascal to Von Neumann*. Princeton University Press. pp. 266–267. ISBN 0-691-08104-2.

[8] Taub, Abraham (1963). *John von Neumann Collected Works* **5**. Macmillan. pp. 80–151.

[9] Akera, Atsushi; Frederik Nebeker (2002). *From 0 to 1: An Authoritative History of Modern Computing*. Oxford University Press. p. 105. ISBN 0-19-514025-7.

[10] Bohl, Rynn: “Tools for Structured and Object-Oriented Design”, Prentice Hall, 2007.

[11] Alan B. Sterneckert (2003) *Critical Incident Management*. p. 126

[12] Andrew Veronis (1978) *Microprocessors: Design and Applications*. p. 111

[13] Marilyn Bohl (1978) *A Guide for Programmers*. p. 65.

[14] Mark A. Fryman (2001) *Quality and Process Improvement*. p. 169.

9.8 Further reading

- ISO (1985). *Information processing -- Documentation symbols and conventions for data, program and system flowcharts, program network charts and system resources charts*. International Organization for Standardization. ISO 5807:1985.

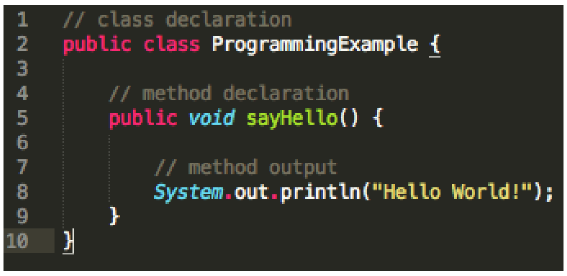
- ISO 10628: Flow Diagrams For Process Plants - General Rules
- ECMA 4: Flowcharts (withdrawn - list of withdrawn standards)
- Schultheiss, Louis A., and Edward M. Heiliger. "Techniques of flow-charting.” (1963); with introduction by Edward Heiliger.

9.9 External links

- Flowcharting Techniques An IBM manual from 1969 (5MB PDF format)
- Advanced Flowchart - Why and how to create advanced flowchart

Chapter 10

Programming language



```
// class declaration
public class ProgrammingExample {

    // method declaration
    public void sayHello() {

        // method output
        System.out.println("Hello World!");
    }
}
```

An example of source code written in the Java programming language, which will print the message "Hello World!" to the standard output when it is compiled and then run by the Java Virtual Machine.

A **programming language** is a formal constructed language designed to communicate instructions to a machine, particularly a computer. Programming languages can be used to create programs to control the behavior of a machine or to express algorithms.

The earliest programming languages preceded the invention of the digital computer and were used to direct the behavior of machines such as Jacquard looms and player pianos.[1] Thousands of different programming languages have been created, mainly in the computer field, and many more still are being created every year. Many programming languages require computation to be specified in an imperative form (i.e., as a sequence of operations to perform), while other languages utilize other forms of program specification such as the declarative form (i.e. the desired result is specified, not how to achieve it).

The description of a programming language is usually split into the two components of syntax (form) and semantics (meaning). Some languages are defined by a specification document (for example, the C programming language is specified by an ISO Standard), while other languages (such as Perl) have a dominant implementation that is treated as a reference.

10.1 Definitions

A programming language is a notation for writing programs, which are specifications of a computation or algorithm.[2] Some, but not all, authors restrict the term "programming language" to those languages that can express *all* possible algorithms.[2][3] Traits often considered important for what constitutes a programming language include:

Function and target A *computer programming language* is a language used to write computer programs, which involve a computer performing some kind of computation[4] or algorithm and possibly control external devices such as printers, disk drives, robots,[5] and so on. For example, PostScript programs are frequently created by another program to control a computer printer or display. More generally, a programming language may describe computation on some, possibly abstract, machine. It is generally accepted that a complete specification for a programming language includes a description, possibly idealized, of a machine or processor for that language.[6] In most practical contexts, a programming language involves a computer; consequently, programming languages are usually defined and studied this way.[7] Programming languages differ from natural languages in that natural languages are only used for interaction between people, while programming languages also allow humans to communicate instructions to machines.

Abstractions Programming languages usually contain abstractions for defining and manipulating data structures or controlling the flow of execution. The practical necessity that a programming language support adequate abstractions is expressed by the abstraction principle;[8] this principle is sometimes formulated as recommendation to the programmer to make proper use of such abstractions.[9]

Expressive power The theory of computation classifies languages by the computations they are capable of expressing. All Turing complete languages can implement the same set of algorithms. ANSI/ISO SQL-92 and Charity are examples of languages that are not Turing complete, yet often called programming languages.[10][11]

Markup languages like XML, HTML or troff, which define structured data, are not usually considered programming languages.[12][13][14] Programming languages may, however, share the syntax with markup languages if a computational semantics is defined. XSLT, for example, is a Turing complete XML dialect.[15][16][17] Moreover, LaTeX, which is mostly used for structuring documents, also contains a Turing complete subset.[18][19]

The term *computer language* is sometimes used interchangeably with programming language.[20] However, the usage of both terms varies among authors, including the exact scope of each. One usage describes programming languages as a subset of computer languages.[21] In this vein, languages used in computing that have a different goal than expressing computer programs are generically designated computer languages. For instance, markup languages are sometimes referred to as computer languages to emphasize that they are not meant to be used for programming.[22]

Another usage regards programming languages as theoretical constructs for programming abstract machines, and computer languages as the subset thereof that runs on physical computers, which have finite hardware resources.[23] John C. Reynolds emphasizes that formal specification languages are just as much programming languages as are the languages intended for execution. He also argues that textual and even graphical input formats that affect the behavior of a computer are programming languages, despite the fact they are commonly not Turing-complete, and remarks that ignorance of programming language concepts is the reason for many flaws in input formats.[24]

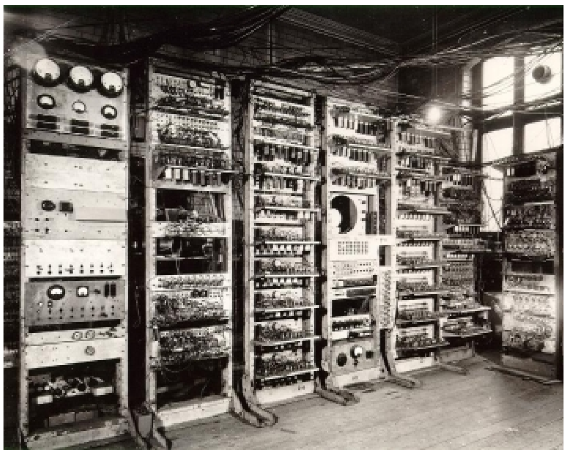
10.2 History

Main articles: History of programming languages and Programming language generations

10.2.1 Early developments

The first programming languages designed to communicate instructions to a computer were written in the 1950s. An early high-level programming language to be designed for a computer was Plankalkül, developed for the German Z3 by Konrad Zuse between 1943 and 1945. However, it was not implemented until 1998 and 2000.[25]

John Mauchly's Short Code, proposed in 1949, was one of the first high-level languages ever developed for an electronic computer.[26] Unlike machine code, Short Code statements represented mathematical expressions in understandable form. However, the program had to be translated into machine code every time it ran, making the process much slower than running the equivalent machine code.



The Manchester Mark 1 ran programs written in Autocode from 1952.

At the University of Manchester, Alick Glennie developed Autocode in the early 1950s. A programming language, it used a compiler to automatically convert the language into machine code. The first code and compiler was developed in 1952 for the Mark 1 computer at the University of Manchester and is considered to be the first compiled high-level programming language.[27][28]

The second autocode was developed for the Mark 1 by R. A. Brooker in 1954 and was called the "Mark 1 Autocode". Brooker also developed an autocode for the Ferranti Mercury in the 1950s in conjunction with the University of Manchester. The version for the EDSAC 2 was devised by D. F. Hartley of University of Cambridge Mathematical Laboratory in 1961. Known as EDSAC 2 Autocode, it was a straight development from Mercury Autocode adapted for local circumstances, and was noted for its object code optimisation and source-language diagnostics which were advanced for the time. A contemporary but separate thread of development, Atlas Autocode was developed for the University of Manchester Atlas 1 machine.

Another early programming language was devised by Grace Hopper in the US, called FLOW-MATIC. It was developed for the UNIVAC I at Remington Rand during the period from 1955 until 1959. Hopper found that business data processing customers were uncomfortable with mathematical notation, and in early 1955, she and her team wrote a specification for an English programming language and implemented a prototype.[29] The FLOW-MATIC compiler became publicly available in early 1958 and was substantially complete in 1959.[30] Flow-Matic was a major influence in the design of COBOL, since only it and its direct descendent AIMACO were in actual use at the time.[31] The language Fortran was developed at IBM in the mid '50s, and became the first widely used high-level general purpose programming language.

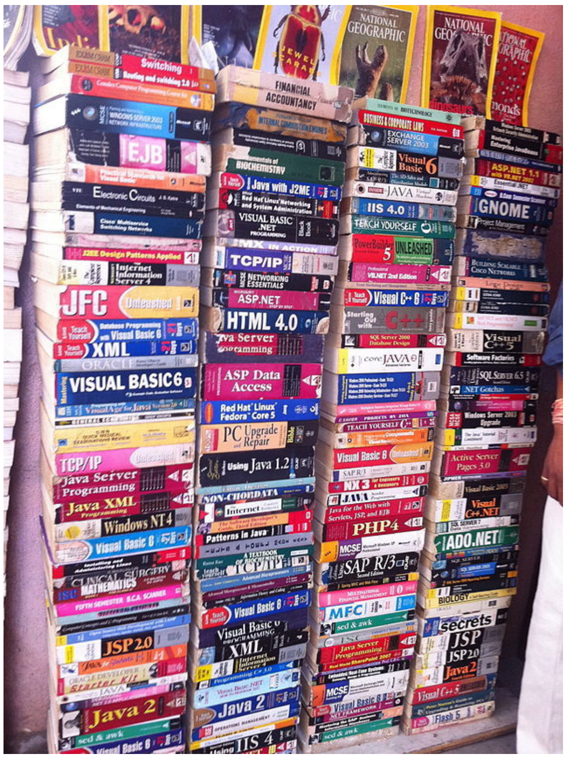
10.2.2 Refinement

The period from the 1960s to the late 1970s brought the development of the major language paradigms now in use:

- APL introduced *array programming* and influenced functional programming.[32]
- ALGOL refined both *structured procedural programming* and the discipline of language specification; the “Revised Report on the Algorithmic Language ALGOL 60" became a model for how later language specifications were written.
- In the 1960s, Simula was the first language designed to support *object-oriented programming*; in the mid-1970s, Smalltalk followed with the first “purely” object-oriented language.
- C was developed between 1969 and 1973 as a *system programming language*, and remains popular.[33]
- Prolog, designed in 1972, was the first *logic programming* language.
- In 1978, ML built a polymorphic type system on top of Lisp, pioneering *statically typed functional programming* languages.

Each of these languages spawned descendants, and most modern programming languages count at least one of them in their ancestry.

The 1960s and 1970s also saw considerable debate over the merits of *structured programming*, and whether programming languages should be designed to support it.[34] Edsger Dijkstra, in a famous 1968 letter published in the Communications of the ACM, argued that GOTO statements should be eliminated from all “higher level” programming languages.[35]


A selection of textbooks that teach programming, in languages both popular and obscure. These are only a few of the thousands of programming languages and dialects that have been designed in history.

10.2.3 Consolidation and growth

The 1980s were years of relative consolidation. C++ combined object-oriented and systems programming. The United States government standardized Ada, a systems programming language derived from Pascal and intended for use by defense contractors. In Japan and elsewhere, vast sums were spent investigating so-called “fifth generation” languages that incorporated logic programming constructs.[36] The functional languages community moved to standardize ML and Lisp. Rather than inventing new paradigms, all of these movements elaborated upon the ideas invented in the previous decade.

One important trend in language design for programming large-scale systems during the 1980s was an increased focus on the use of *modules*, or large-scale organizational units of code. Modula-2, Ada, and ML all developed notable module systems in the 1980s, although other languages, such as PL/I, already had extensive support for modular programming. Module systems were often wedded to generic programming constructs.[37]

The rapid growth of the Internet in the mid-1990s created opportunities for new languages. Perl, originally a Unix scripting tool first released in 1987, became common in dynamic websites. Java came to be used for server-side programming, and bytecode virtual machines became popular again in commercial settings with their promise of "Write once, run anywhere" (UCSD Pascal had been popular for a time in the early 1980s). These developments were not fundamentally novel, rather they were refinements to existing languages and paradigms, and largely based on the C family of programming languages.

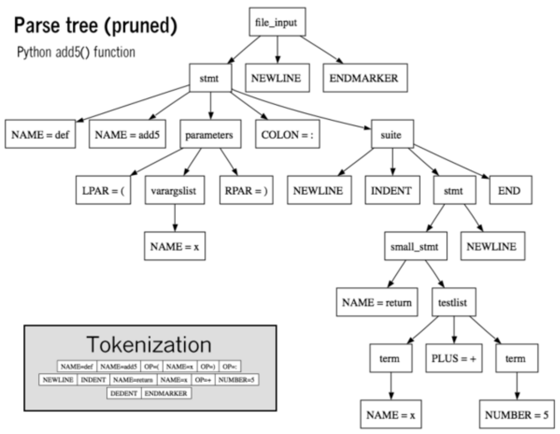
Programming language evolution continues, in both industry and research. Current directions include security and reliability verification, new kinds of modularity (mixins, delegates, aspects), and database integration such as Microsoft’s LINQ.

The 4GLs are examples of languages which are domain-specific, such as SQL, which manipulates and returns sets of data rather than the scalar values which are canonical to most programming languages. Perl, for example, with its "here document" can hold multiple 4GL programs, as well as multiple JavaScript programs, in part of its own perl code and use variable interpolation in the “here document” to support multi-language programming.[38]

10.3 Elements

All programming languages have some primitive building blocks for the description of data and the processes or transformations applied to them (like the addition of two numbers or the selection of an item from a collection). These primitives are defined by syntactic and semantic rules which describe their structure and meaning respectively.

10.3.1 Syntax



Parse tree of Python code with inset tokenization

```
def add5(x):
    return x+5

def dotwrite(ast):
    nodename = getNodename()
    label=symbol.sym_name.get(int(ast[0]),ast[0])
    print '    %s [label="%s' % (nodename, label),
    if isinstance(ast[1], str):
        if ast[1].strip():
            print '= %s"];' % ast[1]
        else:
            print '"]'
    else:
        print '"];'
        children = []
        for n, child in enumerate(ast[1:]):
            children.append(dotwrite(child))
        print '    %s -> {' % nodename,
        for name in children:
            print '%s' % name,
```

Syntax highlighting is often used to aid programmers in recognizing elements of source code. The language above is Python.

Main article: Syntax (programming languages)

A programming language's surface form is known as its syntax. Most programming languages are purely textual; they use sequences of text including words, numbers, and punctuation, much like written natural languages. On the other hand, there are some programming languages which are more graphical in nature, using visual relationships between symbols to specify a program.

The syntax of a language describes the possible combinations of symbols that form a syntactically correct program. The meaning given to a combination of symbols is handled by semantics (either formal or hard-coded in a reference implementation). Since most languages are textual, this article discusses textual syntax.

Programming language syntax is usually defined using a combination of regular expressions (for lexical structure) and Backus–Naur Form (for grammatical structure). Below is a simple grammar, based on Lisp:

expression ::= atom | list atom ::= number | symbol number ::= [+-]?['0'-'9']+ symbol ::= ['A'-'Z''a'-'z'].* list ::= '(' expression* ')'

This grammar specifies the following:

- an *expression* is either an *atom* or a *list*;
- an *atom* is either a *number* or a *symbol*;
- a *number* is an unbroken sequence of one or more decimal digits, optionally preceded by a plus or minus sign;
- a *symbol* is a letter followed by zero or more of any characters (excluding whitespace); and
- a *list* is a matched pair of parentheses, with zero or more *expressions* inside it.

The following are examples of well-formed token sequences in this grammar: 12345, () and (a b c232 (1)).

Not all syntactically correct programs are semantically correct. Many syntactically correct programs are nonetheless ill-formed, per the language's rules; and may (depending on the language specification and the soundness of the implementation) result in an error on translation or execution. In some cases, such programs may exhibit undefined behavior. Even when a program is well-defined within a language, it may still have a meaning that is not intended by the person who wrote it.

Using natural language as an example, it may not be possible to assign a meaning to a grammatically correct sentence or the sentence may be false:

- "Colorless green ideas sleep furiously." is grammatically well-formed but has no generally accepted meaning.
- "John is a married bachelor." is grammatically well-formed but expresses a meaning that cannot be true.

The following C language fragment is syntactically correct, but performs operations that are not semantically defined (the operation *p >> 4 has no meaning for a value having a complex type and p->im is not defined because the value of p is the null pointer):

complex *p = NULL; complex abs_p = sqrt(*p >> 4 + p->im);

If the type declaration on the first line were omitted, the program would trigger an error on compilation, as the variable “p” would not be defined. But the program would still be syntactically correct, since type declarations provide only semantic information.

The grammar needed to specify a programming language can be classified by its position in the Chomsky hierarchy. The syntax of most programming languages can be specified using a Type-2 grammar, i.e., they are context-free grammars.[39] Some languages, including Perl and Lisp, contain constructs that allow execution during the parsing phase. Languages that have constructs that allow the programmer to alter the behavior of the parser make syntax analysis an undecidable problem, and generally blur the distinction between parsing and execution.[40] In contrast to Lisp’s macro system and Perl’s BEGIN blocks, which may contain general computations, C macros are merely string replacements, and do not require code execution.[41]

10.3.2 Semantics

The term Semantics refers to the meaning of languages, as opposed to their form (syntax).

Static semantics

The static semantics defines restrictions on the structure of valid texts that are hard or impossible to express in standard syntactic formalisms.[2] For compiled languages, static semantics essentially include those semantic rules that can be checked at compile time. Examples include checking that every identifier is declared before it is used (in languages that require such declarations) or that the labels on the arms of a case statement are distinct.[42] Many important restrictions of this type, like checking that identifiers are used in the appropriate context (e.g. not adding an integer to a function name), or that subroutine calls have the appropriate number and type of arguments, can be enforced by defining them as rules in a logic called a type system. Other forms of static analyses like data flow analysis may also be part of static semantics. Newer programming languages like Java and C# have definite assignment analysis, a form of data flow analysis, as part of their static semantics.

Dynamic semantics

Main article: Semantics of programming languages

Once data has been specified, the machine must be instructed to perform operations on the data. For example, the semantics may define the strategy by which expressions are evaluated to values, or the manner in which control structures conditionally execute statements. The *dynamic semantics* (also known as *execution semantics*) of a language defines how and when the various constructs of a language should produce a program behavior. There are many ways of defining execution semantics. Natural language is often used to specify the execution semantics of languages commonly used in practice. A significant amount of academic research went into formal semantics of programming languages, which allow execution semantics to be specified in a formal manner. Results from this field of research have seen limited application to programming language design and implementation outside academia.

Type system

Main articles: Data type, Type system and Type safety

A type system defines how a programming language classifies values and expressions into *types*, how it can manipulate those types and how they interact. The goal of a type system is to verify and usually enforce a certain level of correctness in programs written in that language by detecting certain incorrect operations. Any decidable type system involves a trade-off: while it rejects many incorrect programs, it can also prohibit some correct, albeit unusual programs. In order to bypass this downside, a number of languages have *type loopholes*, usually unchecked casts that may be used by the programmer to explicitly allow a normally disallowed operation between different types. In most typed languages, the type system is used only to type check programs, but a number of languages, usually functional ones, infer types, relieving the programmer from the need to write type annotations. The formal design and study of type systems is known as *type theory*.

Typed versus untyped languages A language is *typed* if the specification of every operation defines types of data to which the operation is applicable, with the implication that it is not applicable to other types.[43] For example, the data represented by “this text between the quotes” is a string, and in many programming languages dividing a number by a string has no meaning and will be rejected by the compilers. The invalid operation may be detected when the program is compiled (“static” type checking) and will be rejected by the compiler with a compilation error message, or it may be detected when the program is run (“dynamic” type checking), resulting in a run-time exception. Many languages allow a function called an exception handler to be written to handle this exception and, for example, always return "-1" as the result.

A special case of typed languages are the *single-type* languages. These are often scripting or markup languages, such as REXX or SGML, and have only one data type—most commonly character strings which are used for both symbolic and numeric data.

In contrast, an *untyped language*, such as most assembly languages, allows any operation to be performed on any data, which are generally considered to be sequences of bits of various lengths.[43] High-level languages which are untyped include BCPL, Tcl, and some varieties of Forth.

In practice, while few languages are considered typed from the point of view of type theory (verifying or rejecting *all* operations), most modern languages offer a degree of typing.[43] Many production languages provide means to bypass or subvert the type system, trading type-safety for finer control over the program's execution (see casting).

Static versus dynamic typing In *static typing*, all expressions have their types determined prior to when the program is executed, typically at compile-time. For example, 1 and (2+2) are integer expressions; they cannot be passed to a function that expects a string, or stored in a variable that is defined to hold dates.[43]

Statically typed languages can be either *manifestly typed* or *type-inferred*. In the first case, the programmer must explicitly write types at certain textual positions (for example, at variable declarations). In the second case, the compiler *infers* the types of expressions and declarations based on context. Most mainstream statically typed languages, such as C++, C# and Java, are manifestly typed. Complete type inference has traditionally been associated with less mainstream languages, such as Haskell and ML. However, many manifestly typed languages support partial type inference; for example, Java and C# both infer types in certain limited cases.[44]

Dynamic typing, also called *latent typing*, determines the type-safety of operations at run time; in other words, types are associated with *run-time values* rather than *textual expressions*.[43] As with type-inferred languages, dynamically typed languages do not require the programmer to write explicit type annotations on expressions. Among other things, this may permit a single variable to refer to values of different types at different points in the program execution. However, type errors cannot be automatically detected until a piece of code is actually executed, potentially making debugging more difficult. Lisp, Perl, Python, JavaScript, and Ruby are dynamically typed.

Weak and strong typing *Weak typing* allows a value of one type to be treated as another, for example treating a string as a number.[43] This can occasionally be useful, but it can also allow some kinds of program faults to go undetected at compile time and even at run time.

Strong typing prevents the above. An attempt to perform an operation on the wrong type of value raises an error.[43] Strongly typed languages are often termed *type-safe* or *safe*.

An alternative definition for "weakly typed" refers to languages, such as Perl and JavaScript, which permit a large number of implicit type conversions. In JavaScript, for example, the expression 2 * x implicitly converts x to a number, and this conversion succeeds even if x is null, undefined, an Array, or a string of letters. Such implicit conversions are often useful, but they can mask programming errors. *Strong* and *static* are now generally considered orthogonal concepts, but usage in the literature differs. Some use the term *strongly typed* to mean *strongly, statically typed*, or, even more confusingly, to mean simply *statically typed*. Thus C has been called both strongly typed and weakly, statically typed.[45][46]

It may seem odd to some professional programmers that C could be "weakly, statically typed". However, notice that the use of the generic pointer, the **void*** pointer, does allow for casting of pointers to other pointers without needing to do an explicit cast. This is extremely similar to somehow casting an array of bytes to any kind of datatype in C without using an explicit cast, such as (int) or (char).

10.3.3 Standard library and run-time system

Main article: Standard library

Most programming languages have an associated core library (sometimes known as the 'standard library', especially if it is included as part of the published language standard), which is conventionally made available by all implementations of the language. Core libraries typically include definitions for commonly used algorithms, data structures, and mechanisms for input and output.

The line between a language and its core library differs from language to language. In some cases, the language designers may treat the library as a separate entity from the language. However, a language's core library is often treated as part of the language by its users, and some language specifications even require that this library be made available in all implementations. Indeed, some languages are designed so that the meanings of certain syntactic constructs cannot even be described without referring to the core library. For example, in Java, a string literal is defined as an instance of the java.lang.String class; similarly, in Smalltalk, an anonymous function expression (a "block") constructs an instance of the library's BlockContext class. Conversely, Scheme contains multiple coherent subsets that suffice to construct the rest of the language as library macros, and so the language designers do not even bother to say which portions of the language must be implemented as language constructs, and which must be implemented as parts of a library.

10.4 Design and implementation

Programming languages share properties with natural languages related to their purpose as vehicles for communication, having a syntactic form separate from its semantics, and showing *language families* of related languages branching one from another.[47][48] But as artificial constructs, they also differ in fundamental ways from languages that have evolved through usage. A significant difference is that a programming language can be fully described and studied in its entirety, since it has a precise and finite definition.[49] By contrast, natural languages have changing meanings given by their users in different communities. While constructed languages are also artificial languages designed from the ground up with a specific purpose, they lack the precise and complete semantic definition that a programming language has.

Many programming languages have been designed from scratch, altered to meet new needs, and combined with other languages. Many have eventually fallen into disuse. Although there have been attempts to design one "universal" programming language that serves all purposes, all of them have failed to be generally accepted as filling this role.[50] The need for diverse programming languages arises from the diversity of contexts in which languages are used:

- Programs range from tiny scripts written by individual hobbyists to huge systems written by hundreds of programmers.
- Programmers range in expertise from novices who need simplicity above all else, to experts who may be comfortable with considerable complexity.
- Programs must balance speed, size, and simplicity on systems ranging from microcontrollers to supercomputers.
- Programs may be written once and not change for generations, or they may undergo continual modification.
- Programmers may simply differ in their tastes: they may be accustomed to discussing problems and expressing them in a particular language.

One common trend in the development of programming languages has been to add more ability to solve problems using a higher level of abstraction. The earliest programming languages were tied very closely to the underlying hardware of the computer. As new programming languages have developed, features have been added that let programmers express ideas that are more remote from simple translation into underlying hardware instructions. Because programmers are less tied to the complexity of the computer, their programs can do more computing with less effort from the programmer. This lets them write more functionality per time unit.[51]

Natural language programming has been proposed as a way to eliminate the need for a specialized language for programming. However, this goal remains distant and its benefits are open to debate. Edsger W. Dijkstra took the position that the use of a formal language is essential to prevent the introduction of meaningless constructs, and dismissed natural language programming as "foolish".[52] Alan Perlis was similarly dismissive of the idea.[53] Hybrid approaches have been taken in Structured English and SQL.

A language's designers and users must construct a number of artifacts that govern and enable the practice of programming. The most important of these artifacts are the language *specification* and *implementation*.

10.4.1 Specification

Main article: Programming language specification

The specification of a programming language is an artifact that the language users and the implementors can use to agree upon whether a piece of source code is a valid program in that language, and if so what its behavior shall be.

A programming language specification can take several forms, including the following:

- An explicit definition of the syntax, static semantics, and execution semantics of the language. While syntax is commonly specified using a formal grammar, semantic definitions may be written in natural language (e.g., as in the C language), or a formal semantics (e.g., as in Standard ML[54] and Scheme[55] specifications).
- A description of the behavior of a translator for the language (e.g., the C++ and Fortran specifications). The syntax and semantics of the language have to be inferred from this description, which may be written in natural or a formal language.
- A *reference* or *model* implementation, sometimes written in the language being specified (e.g., Prolog or ANSI REXX[56]). The syntax and semantics of the language are explicit in the behavior of the reference implementation.

10.4.2 Implementation

Main article: Programming language implementation

An *implementation* of a programming language provides a way to write programs in that language and execute them on one or more configurations of hardware and software. There are, broadly, two approaches to programming language implementation: *compilation* and *interpretation*. It is generally possible to implement a language using either technique.

The output of a compiler may be executed by hardware or a program called an interpreter. In some implementations that make use of the interpreter approach there is no distinct boundary between compiling and interpreting. For instance, some implementations of BASIC compile and then execute the source a line at a time.

Programs that are executed directly on the hardware usually run several orders of magnitude faster than those that are interpreted in software.

One technique for improving the performance of interpreted programs is just-in-time compilation. Here the virtual machine, just before execution, translates the blocks of bytecode which are going to be used to machine code, for direct execution on the hardware.

10.5 Usage

Thousands of different programming languages have been created, mainly in the computing field.[57]

Programming languages differ from most other forms of human expression in that they require a greater degree of precision and completeness. When using a natural language to communicate with other people, human authors and speakers can be ambiguous and make small errors, and still expect their intent to be understood. However, figuratively speaking, computers "do exactly what they are told to do", and cannot "understand" what code the programmer intended to write. The combination of the language definition, a program, and the program's inputs must fully specify the external behavior that occurs when the program is executed, within the domain of control of that program. On the other hand, ideas about an algorithm can be communicated to humans without the precision required for execution by using pseudocode, which interleaves natural language with code written in a programming language.

A programming language provides a structured mechanism for defining pieces of data, and the operations or transformations that may be carried out automatically on that data. A programmer uses the abstractions present in the language to represent the concepts involved in a computation. These concepts are represented as a collection of the simplest elements available (called primitives).[58] *Programming* is the process by which programmers combine these primitives to compose new programs, or adapt existing ones to new uses or a changing environment.

Programs for a computer might be executed in a batch process without human interaction, or a user might type commands in an interactive session of an interpreter. In this case the "commands" are simply programs, whose execution is chained together. When a language can run its commands through an interpreter (such as a Unix shell or other command-line interface), without compiling, it is called a scripting language.[59]

10.5.1 Measuring language usage

Main article: Measuring programming language popularity

It is difficult to determine which programming languages are most widely used, and what usage means varies by context. One language may occupy the greater number of programmer hours, a different one have more lines of code, and a third utilize the most CPU time. Some languages are very popular for particular kinds of applications. For example, COBOL is still strong in the corporate data center, often on large mainframes;[60][61] Fortran in scientific and engineering applications; Ada in aerospace, transportation, military, real-time and embedded applications; and C in embedded applications and operating systems. Other languages are regularly used to write many different kinds of applications.

Various methods of measuring language popularity, each subject to a different bias over what is measured, have been proposed:

- counting the number of job advertisements that mention the language[62]
- the number of books sold that teach or describe the language[63]
- estimates of the number of existing lines of code written in the language – which may underestimate languages not often found in public searches[64]
- counts of language references (i.e., to the name of the language) found using a web search engine.

Combining and averaging information from various internet sites, langpop.com claims that in 2013 the ten most popular programming languages are (in descending order by overall popularity): C, Java, PHP, JavaScript, C++, Python, Shell, Ruby, Objective-C and C#.[65]

10.6 Taxonomies

For more details on this topic, see Categorical list of programming languages.

There is no overarching classification scheme for programming languages. A given programming language does not usually have a single ancestor language. Languages commonly arise by combining the elements of several predecessor languages with new ideas in circulation at the time. Ideas that originate in one language will diffuse throughout a family of related languages, and then leap suddenly across familial gaps to appear in an entirely different family.

The task is further complicated by the fact that languages can be classified along multiple axes. For example, Java is both an object-oriented language (because it encourages object-oriented organization) and a concurrent language (because it contains built-in constructs for running multiple threads in parallel). Python is an object-oriented scripting language.

In broad strokes, programming languages divide into *programming paradigms* and a classification by *intended domain of use,* with general-purpose programming languages distinguished from domain-specific programming languages. Traditionally, programming languages have been regarded as describing computation in terms of imperative sentences, i.e. issuing commands. These are generally called imperative programming languages. A great deal of research in programming languages has been aimed at blurring the distinction between a program as a set of instructions and a program as an assertion about the desired answer, which is the main feature of declarative programming.[66] More refined paradigms include procedural programming, object-oriented programming, functional programming, and logic programming; some languages are hybrids of paradigms or multi-paradigmatic. An assembly language is not so much a paradigm as a direct model of an underlying machine architecture. By purpose, programming languages might be considered general purpose, system programming languages, scripting languages, domain-specific languages, or concurrent/distributed languages (or a combination of these).[67] Some general purpose languages were designed largely with educational goals.[68]

A programming language may also be classified by factors unrelated to programming paradigm. For instance, most programming languages use English language keywords, while a minority do not. Other languages may be classified as being deliberately esoteric or not.

10.7 See also

- Comparison of programming languages (basic instructions)
- Comparison of programming languages
- Computer programming
- Computer science and Outline of computer science
- Educational programming language
- Invariant based programming
- Lists of programming languages
- List of programming language researchers
- Programming languages used in most popular websites
- Literate programming
- Dialect (computing)
- Programming language theory
- Pseudocode
- Scientific language
- Software engineering and List of software engineering topics

10.8 References

[1] Ettinger, James (2004) *Jacquard's Web*, Oxford University Press

[2] Aaby, Anthony (2004). *Introduction to Programming Languages*.

[3] In mathematical terms, this means the programming language is Turing-complete MacLennan, Bruce J. (1987). *Principles of Programming Languages*. Oxford University Press. p. 1. ISBN 0-19-511306-3.

[4] ACM SIGPLAN (2003). "Bylaws of the Special Interest Group on Programming Languages of the Association for Computing Machinery". Retrieved 19 June 2006., *The scope of SIGPLAN is the theory, design, implementation, description, and application of computer programming languages - languages that permit the specification of a variety of different computations, thereby providing the user with significant control (immediate or delayed) over the computer's operation.*

[5] Dean, Tom (2002). "Programming Robots". *Building Intelligent Robots*. Brown University Department of Computer Science. Retrieved 23 September 2006.

[6] R. Narasimahan, Programming Languages and Computers: A Unified Metatheory, pp. 189-—247 in Franz Alt, Morris Rubinoff (eds.) Advances in computers, Volume 8, Academic Press, 1994, ISBN 0-12-012108-5, p.193 : "a complete specification of a programming language must, by definition, include a specification of a processor--idealized, if you will--for that language." [the source cites many references to support this statement]

[7] Ben Ari, Mordechai (1996). *Understanding Programming Languages*. John Wiley and Sons. Programs and languages can be defined as purely formal mathematical objects. However, more people are interested in programs than in other mathematical objects such as groups, precisely because it is possible to use the program—the sequence of symbols—to control the execution of a computer. While we highly recommend the study of the theory of programming, this text will generally limit itself to the study of programs as they are executed on a computer.

[8] David A. Schmidt, *The structure of typed programming languages*, MIT Press, 1994, ISBN 0-262-19349-3, p. 32

[9] Pierce, Benjamin (2002). *Types and Programming Languages*. MIT Press. p. 339. ISBN 0-262-16209-1.

[10] Digital Equipment Corporation. "Information Technology - Database Language SQL (Proposed revised text of DIS 9075)". *ISO/IEC 9075:1992, Database Language SQL*. Retrieved 29 June 2006.

[11] The Charity Development Group (December 1996). "The CHARITY Home Page". Retrieved 29 June 2006., *Charity is a categorical programming language...*, *All Charity computations terminate.*

[12] XML in 10 points W3C, 1999, *XML is not a programming language.*

[13] Powell, Thomas (2003). *HTML & XHTML: the complete reference*. McGraw-Hill. p. 25. ISBN 0-07-222942-X. *HTML is not a programming language.*

[14] Dykes, Lucinda; Tittel, Ed (2005). *XML For Dummies, 4th Edition*. Wiley. p. 20. ISBN 0-7645-8845-1. *...it's a markup language, not a programming language.*

[15] "What kind of language is XSLT?". IBM.com. Retrieved 3 December 2010.

[16] "XSLT is a Programming Language". Msdn.microsoft.com. Retrieved 3 December 2010.

[17] Scott, Michael (2006). *Programming Language Pragmatics*. Morgan Kaufmann. p. 802. ISBN 0-12-633951-1. *XSLT, though highly specialized to the transformation of XML, is a Turing-complete programming language.*

[18] http://tobi.oetiker.ch/lshort/lshort.pdf

[19] Syropoulos, Apostolos; Antonis Tsolomitis; Nick Sofroniou (2003). *Digital typography using LaTeX*. Springer-Verlag. p. 213. ISBN 0-387-95217-9. *TeX is not only an excellent typesetting engine but also a real programming language.*

[20] Robert A. Edmunds, The Prentice-Hall standard glossary of computer terminology, Prentice-Hall, 1985, p. 91

[21] Pascal Lando, Anne Lapujade, Gilles Kassel, and Frédéric Fürst, *Towards a General Ontology of Computer Programs*, ICSOFT 2007, pp. 163-170

[22] S.K. Bajpai, *Introduction To Computers And C Programming*, New Age International, 2007, ISBN 81-224-1379-X, p. 346

[23] R. Narasimahan, Programming Languages and Computers: A Unified Metatheory, pp. 189-—247 in Franz Alt, Morris Rubinoff (eds.) Advances in computers, Volume 8, Academic Press, 1994, ISBN 0-12-012108-5, p.215: "[...] the model [...] for computer languages differs from that [...] for programming languages in only two respects. In a computer language, there are only finitely many names--or registers--which can assume only finitely many values--or states--and these states are not further distinguished in terms of any other attributes. [author's footnote:] This may sound like a truism but its implications are far reaching. For example, it would imply that any model for programming languages, by fixing certain of its parameters or features, should be reducible in a natural way to a model for computer languages."

[24] John C. Reynolds, *Some thoughts on teaching programming and programming languages*, SIGPLAN Notices, Volume 43, Issue 11, November 2008, p.109

[25] Rojas, Raúl, et al. (2000). "Plankalkül: The First High-Level Programming Language and its Implementation". Institut für Informatik, Freie Universität Berlin, Technical Report B-3/2000. (full text)

[26] Sebesta, W.S Concepts of Programming languages. 2006;M6 14:18 pp.44. ISBN 0-321-33025-0

[27] Knuth, Donald E.; Pardo, Luis Trabb. "Early development of programming languages". *Encyclopedia of Computer Science and Technology* (Marcel Dekker) **7**: 419–493.

[28] Peter J. Bentley (2012). *Digitized: The Science of Computers and how it Shapes Our World*. Oxford University Press. p. 87.

[29] Hopper (1978) p. 16.

[30] Sammet (1969) p. 316

[31] Sammet (1978) p. 204.

[32] Richard L. Wexelblat: *History of Programming Languages*, Academic Press, 1981, chapter XIV.

[33] François Labelle. "Programming Language Usage Graph". *SourceForge*. Retrieved 21 June 2006.. This comparison analyzes trends in number of projects hosted by a popular community programming repository. During most years of the comparison, C leads by a considerable margin; in 2006, Java overtakes C, but the combination of C/C++ still leads considerably.

[34] Hayes, Brian (2006). "The Semicolon Wars". *American Scientist* **94** (4): 299–303. doi:10.1511/2006.60.299.

[35] Dijkstra, Edsger W. (March 1968). "Go To Statement Considered Harmful". *Communications of the ACM* **11** (3): 147–148. doi:10.1145/362929.362947. Retrieved 2014-05-22.

[36] Tetsuro Fujise, Takashi Chikayama, Kazuaki Rokusawa, Akihiko Nakase (December 1994). "KLIC: A Portable Implementation of KL1" *Proc. of FGCS '94, ICOT* Tokyo, December 1994. http://www.icot.or.jp/ARCHIVE/HomePage-E.html KLIC is a portable implementation of a concurrent logic programming language KL1.

[37] Jim Bender (15 March 2004). "Mini-Bibliography on Modules for Functional Programming Languages". *ReadScheme.org*. Retrieved 27 September 2006.

[38] Wall, *Programming Perl* ISBN 0-596-00027-8 p. 66

[39] Michael Sipser (1996). *Introduction to the Theory of Computation*. PWS Publishing. ISBN 0-534-94728-X. Section 2.2: Pushdown Automata, pp.101–114.

[40] Jeffrey Kegler, "Perl and Undecidability", *The Perl Review*. Papers 2 and 3 prove, using respectively Rice's theorem and direct reduction to the halting problem, that the parsing of Perl programs is in general undecidable.

[41] Marty Hall, 1995, Lecture Notes: Macros, PostScript version

[42] Michael Lee Scott, *Programming language pragmatics*, Edition 2, Morgan Kaufmann, 2006, ISBN 0-12-633951-1, p. 18–19

[43] Andrew Cooke. "Introduction To Computer Languages". Retrieved 13 July 2012.

[44] Specifically, instantiations of generic types are inferred for certain expression forms. Type inference in Generic Java—the research language that provided the basis for Java 1.5's bounded parametric polymorphism extensions—is discussed in two informal manuscripts from the Types mailing list: Generic Java type inference is unsound (Alan Jeffrey, 17 December 2001) and Sound Generic Java type inference (Martin Odersky, 15 January 2002). C#'s type system is similar to Java's, and uses a similar partial type inference scheme.

[45] "Revised Report on the Algorithmic Language Scheme". 20 February 1998. Retrieved 9 June 2006.

[46] Luca Cardelli and Peter Wegner. "On Understanding Types, Data Abstraction, and Polymorphism". *Manuscript (1985)*. Retrieved 9 June 2006.

[47] Steven R. Fischer, *A history of language*, Reaktion Books, 2003, ISBN 1-86189-080-X, p. 205

[48] Éric Lévénez (2011). "Computer Languages History".

[49] Jing Huang. "Artificial Language vs. Natural Language".

[50] IBM in first publishing PL/I, for example, rather ambitiously titled its manual *The universal programming language PL/I* (IBM Library; 1966). The title reflected IBM's goals for unlimited subsetting capability: *PL/I is designed in such a way that one can isolate subsets from it satisfying the requirements of particular applications.* ("PL/I". *Encyclopedia of Mathematics*. Retrieved 29 June 2006.). Ada and UNCOL had similar early goals.

[51] Frederick P. Brooks, Jr.: *The Mythical Man-Month,* Addison-Wesley, 1982, pp. 93-94

[52] Dijkstra, Edsger W. On the foolishness of "natural language programming." EWD667.

[53] Perlis, Alan (September 1982). "Epigrams on Programming". *SIGPLAN Notices Vol. 17, No. 9*. pp. 7–13.

[54] Milner, R.; M. Tofte, R. Harper and D. MacQueen. (1997). *The Definition of Standard ML (Revised)*. MIT Press. ISBN 0-262-63181-4.

[55] Kelsey, Richard; William Clinger and Jonathan Rees (February 1998). "Section 7.2 Formal semantics". *Revised^5 Report on the Algorithmic Language Scheme*. Retrieved 9 June 2006.

[56] ANSI — Programming Language Rexx, X3-274.1996

[57] "HOPL: an interactive Roster of Programming Languages". Australia: Murdoch University. Retrieved 1 June 2009. This site lists 8512 languages.

[58] Abelson, Sussman, and Sussman. "Structure and Interpretation of Computer Programs". Retrieved 3 March 2009.

[59] Brown Vicki (1999). "Scripting Languages". *mactech.com*. Retrieved November 17, 2014.

[60] Georgina Swan (2009-09-21). "COBOL turns 50". computerworld.com.au. Retrieved 2013-10-19.

[61] Ed Airey (2012-05-03). "7 Myths of COBOL Debunked". developer.com. Retrieved 2013-10-19.

[62] Nicholas Enticknap. "SSL/Computer Weekly IT salary survey: finance boom drives IT job growth". Computerweekly.com. Retrieved 2013-06-14.

[63] "Counting programming languages by book sales". Radar.oreilly.com. 2 August 2006. Retrieved 3 December 2010.

[64] Bieman, J.M.; Murdock, V., Finding code on the World Wide Web: a preliminary investigation, Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation, 2001

[65] "Programming Language Popularity". langpop.com. 2013-10-25. Retrieved 2014-01-02.

[66] Carl A. Gunter, *Semantics of Programming Languages: Structures and Techniques*, MIT Press, 1992, ISBN 0-262-57095-5, p. 1

[67] "TUNES: Programming Languages".

[68] Wirth, Niklaus (1993). "Recollections about the development of Pascal". *Proc. 2nd ACM SIGPLAN conference on history of programming languages*: 333–342. doi:10.1145/154766.155378. ISBN 0-89791-570-4. Retrieved 30 June 2006.

10.9 Further reading

See also: History of programming languages § Further reading

- Abelson, Harold; Sussman, Gerald Jay (1996). *Structure and Interpretation of Computer Programs* (2nd ed.). MIT Press.
- Raphael Finkel: *Advanced Programming Language Design*, Addison Wesley 1995.
- Daniel P. Friedman, Mitchell Wand, Christopher T. Haynes: *Essentials of Programming Languages*, The MIT Press 2001.
- Maurizio Gabbrielli and Simone Martini: "Programming Languages: Principles and Paradigms", Springer, 2010.
- David Gelernter, Suresh Jagannathan: *Programming Linguistics*, The MIT Press 1990.

- Ellis Horowitz (ed.): *Programming Languages, a Grand Tour* (3rd ed.), 1987.
- Ellis Horowitz: *Fundamentals of Programming Languages*, 1989.
- Shriram Krishnamurthi: *Programming Languages: Application and Interpretation*, online publication.
- Bruce J. MacLennan: *Principles of Programming Languages: Design, Evaluation, and Implementation*, Oxford University Press 1999.
- John C. Mitchell: *Concepts in Programming Languages*, Cambridge University Press 2002.
- Benjamin C. Pierce: *Types and Programming Languages*, The MIT Press 2002.
- Terrence W. Pratt and Marvin V. Zelkowitz: *Programming Languages: Design and Implementation* (4th ed.), Prentice Hall 2000.
- Peter H. Salus. *Handbook of Programming Languages* (4 vols.). Macmillan 1998.
- Ravi Sethi: *Programming Languages: Concepts and Constructs*, 2nd ed., Addison-Wesley 1996.
- Michael L. Scott: *Programming Language Pragmatics*, Morgan Kaufmann Publishers 2005.
- Robert W. Sebesta: *Concepts of Programming Languages*, 9th ed., Addison Wesley 2009.
- Franklyn Turbak and David Gifford with Mark Sheldon: *Design Concepts in Programming Languages*, The MIT Press 2009.
- Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*, The MIT Press 2004.
- David A. Watt. *Programming Language Concepts and Paradigms*. Prentice Hall 1990.
- David A. Watt and Muffy Thomas. *Programming Language Syntax and Semantics*. Prentice Hall 1991.
- David A. Watt. *Programming Language Processors*. Prentice Hall 1993.
- David A. Watt. *Programming Language Design Concepts*. John Wiley & Sons 2004.

10.10 External links

- 99 Bottles of Beer A collection of implementations in many languages.
- Computer Programming Languages at DMOZ

Chapter 11

History of programming languages

This article discusses the major developments in the history of **programming languages**. For a detailed timeline of events, see: Timeline of programming languages.

11.1 Early history

During a nine-month period in 1840-1843, Ada Lovelace translated the memoir of Italian mathematician Luigi Menabrea about Charles Babbage's newest proposed machine, the Analytical Engine. With the article she appended a set of notes which specified in complete detail a method for calculating Bernoulli numbers with the Analytical Engine, recognized by some historians as the world's first computer program.[1]

Herman Hollerith realized that he could encode information on punch cards when he observed that train conductors encode the appearance of the ticket holders on the train tickets using the position of punched holes on the tickets. Hollerith then encoded the 1890 census data on punch cards.

The first computer codes were specialized for their applications. In the first decades of the 20th century, numerical calculations were based on decimal numbers. Eventually it was realized that logic could be represented with numbers, not only with words. For example, Alonzo Church was able to express the lambda calculus in a formulaic way. The Turing machine was an abstraction of the operation of a tape-marking machine, for example, in use at the telephone companies. Turing machines set the basis for storage of programs as data in the von Neumann architecture of computers by representing a machine through a finite number. However, unlike the lambda calculus, Turing's code does not serve well as a basis for higher-level languages—its principal use is in rigorous analyses of algorithmic complexity.

Like many "firsts" in history, the first modern programming language is hard to identify. From the start, the restrictions of the hardware defined the language. Punch cards allowed 80 columns, but some of the columns had to be used for a sorting number on each card. FORTRAN included some keywords which were the same as English words, such as "IF", "GOTO" (go to) and "CONTINUE". The use of a magnetic drum for memory meant that computer programs also had to be interleaved with the rotations of the drum. Thus the programs were more hardware-dependent.

To some people, what was the first modern programming language depends on how much power and human-readability is required before the status of "programming language" is granted. Jacquard looms and Charles Babbage's Difference Engine both had simple, extremely limited languages for describing the actions that these machines should perform. One can even regard the punch holes on a player piano scroll as a limited domain-specific language, albeit not designed for human consumption.

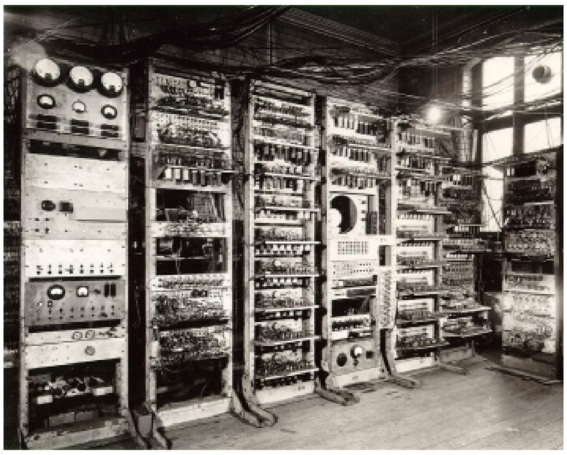
11.2 First programming languages

In the 1940s, the first recognizably modern electrically powered computers were created. The limited speed and memory capacity forced programmers to write hand tuned assembly language programs. It was eventually realized that programming in assembly language required a great deal of intellectual effort and was error-prone.

The first programming languages designed to communicate instructions to a computer were written in the 1950s. An early high-level programming language to be designed for a computer was Plankalkül, developed for the German Z3 by Konrad Zuse between 1943 and 1945. However, it was not implemented until 1998 and 2000.[2]

John Mauchly's Short Code, proposed in 1949, was one of the first high-level languages ever developed for an electronic computer.[3] Unlike machine code, Short Code statements represented mathematical expressions in understandable form. However, the program had to be translated into machine code every time it ran, making the process much slower than running the equivalent machine code.

At the University of Manchester, Alick Glennie developed Autocode in the early 1950s. A programming language, it used a compiler to automatically convert the language into machine code. The first code and compiler was developed in 1952 for the Mark 1 computer at the University of Manchester and is considered to be the first



The Manchester Mark 1 ran programs written in Autocode from 1952.

compiled high-level programming language.[4][5]

The second autocode was developed for the Mark 1 by R. A. Brooker in 1954 and was called the “Mark 1 Autocode”. Brooker also developed an autocode for the Ferranti Mercury in the 1950s in conjunction with the University of Manchester. The version for the EDSAC 2 was devised by D. F. Hartley of University of Cambridge Mathematical Laboratory in 1961. Known as EDSAC 2 Autocode, it was a straight development from Mercury Autocode adapted for local circumstances, and was noted for its object code optimisation and source-language diagnostics which were advanced for the time. A contemporary but separate thread of development, Atlas Autocode was developed for the University of Manchester Atlas 1 machine.

Another early programming language was devised by Grace Hopper in the US, called FLOW-MATIC. It was developed for the UNIVAC I at Remington Rand during the period from 1955 until 1959. Hopper found that business data processing customers were uncomfortable with mathematical notation, and in early 1955, she and her team wrote a specification for an English programming language and implemented a prototype.[6] The FLOW-MATIC compiler became publicly available in early 1958 and was substantially complete in 1959.[7] Flow-Matic was a major influence in the design of COBOL, since only it and its direct descendent AIMACO were in actual use at the time.[8] The language Fortran was developed at IBM in the mid 1950s, and became the first widely used high-level general purpose programming language.

Other languages still in use today, include LISP (1958), invented by John McCarthy and COBOL (1959), created by the Short Range Committee. Another milestone in the late 1950s was the publication, by a committee of American and European computer scientists, of “a new language for algorithms"; the *ALGOL 60 Report* (the "**ALGO**rithmic **L**anguage”). This report consolidated many ideas circulating at the time and featured three key language innovations:

- nested block structure: code sequences and associated declarations could be grouped into blocks without having to be turned into separate, explicitly named procedures;
- lexical scoping: a block could have its own private variables, procedures and functions, invisible to code outside that block, that is, information hiding.

Another innovation, related to this, was in how the language was described:

- a mathematically exact notation, Backus-Naur Form (BNF), was used to describe the language’s syntax. Nearly all subsequent programming languages have used a variant of BNF to describe the context-free portion of their syntax.

Algol 60 was particularly influential in the design of later languages, some of which soon became more popular. The Burroughs large systems were designed to be programmed in an extended subset of Algol.

Algol’s key ideas were continued, producing ALGOL 68:

- syntax and semantics became even more orthogonal, with anonymous routines, a recursive typing system with higher-order functions, etc.;
- not only the context-free part, but the full language syntax and semantics were defined formally, in terms of Van Wijngaarden grammar, a formalism designed specifically for this purpose.

Algol 68’s many little-used language features (for example, concurrent and parallel blocks) and its complex system of syntactic shortcuts and automatic type coercions made it unpopular with implementers and gained it a reputation of being *difficult*. Niklaus Wirth actually walked out of the design committee to create the simpler Pascal language.

Some important languages that were developed in this period include:

11.3 Establishing fundamental paradigms

The period from the late 1960s to the late 1970s brought a major flowering of programming languages. Most of the major language paradigms now in use were invented in this period:

- **Simula**, invented in the late 1960s by Nygaard and Dahl as a superset of Algol 60, was the first language designed to support object-oriented programming.

- **C**, an early systems programming language, was developed by Dennis Ritchie and Ken Thompson at Bell Labs between 1969 and 1973.
- **Smalltalk** (mid-1970s) provided a complete ground-up design of an object-oriented language.
- **Prolog**, designed in 1972 by Colmerauer, Roussel, and Kowalski, was the first logic programming language.
- **ML** built a polymorphic type system (invented by Robin Milner in 1973) on top of Lisp , pioneering statically typed functional programming languages.

Each of these languages spawned an entire family of descendants, and most modern languages count at least one of them in their ancestry.

The 1960s and 1970s also saw considerable debate over the merits of "structured programming", which essentially meant programming without the use of Goto. This debate was closely related to language design: some languages did not include GOTO, which forced structured programming on the programmer. Although the debate raged hotly at the time, nearly all programmers now agree that, even in languages that provide GOTO, it is bad programming style to use it except in rare circumstances. As a result, later generations of language designers have found the structured programming debate tedious and even bewildering.

To provide even faster compile times, some languages were structured for "one-pass compilers" which expect subordinate routines to be defined first, as with Pascal, where the main routine, or driver function, is the final section of the program listing.

Some important languages that were developed in this period include:

11.4 1980s: consolidation, modules, performance

The 1980s were years of relative consolidation in imperative languages. Rather than inventing new paradigms, all of these movements elaborated upon the ideas invented in the previous decade. C++ combined object-oriented and systems programming. The United States government standardized Ada, a systems programming language intended for use by defense contractors. In Japan and elsewhere, vast sums were spent investigating so-called fifth-generation programming languages that incorporated logic programming constructs. The functional languages community moved to standardize ML and Lisp. Research in Miranda, a functional language with lazy evaluation, began to take hold in this decade.

One important new trend in language design was an increased focus on programming for large-scale systems through the use of *modules*, or large-scale organizational units of code. Modula, Ada, and ML all developed notable module systems in the 1980s. Module systems were often wedded to generic programming constructs--generics being, in essence, parametrized modules (see also polymorphism in object-oriented programming).

Although major new paradigms for imperative programming languages did not appear, many researchers expanded on the ideas of prior languages and adapted them to new contexts. For example, the languages of the Argus and Emerald systems adapted object-oriented programming to distributed systems.

The 1980s also brought advances in programming language implementation. The RISC movement in computer architecture postulated that hardware should be designed for compilers rather than for human assembly programmers. Aided by processor speed improvements that enabled increasingly aggressive compilation techniques, the RISC movement sparked greater interest in compilation technology for high-level languages.

Language technology continued along these lines well into the 1990s.

Some important languages that were developed in this period include:

11.5 1990s: the Internet age

The rapid growth of the Internet in the mid-1990s was the next major historic event in programming languages. By opening up a radically new platform for computer systems, the Internet created an opportunity for new languages to be adopted. In particular, the JavaScript programming language rose to popularity because of its early integration with the Netscape Navigator web browser. Various other scripting languages achieved widespread use in developing customized application for web servers such as PHP. The 1990s saw no fundamental novelty in imperative languages, but much recombination and maturation of old ideas. This era began the spread of functional languages. A big driving philosophy was programmer productivity. Many “rapid application development” (RAD) languages emerged, which usually came with an IDE, garbage collection, and were descendants of older languages. All such languages were object-oriented. These included Object Pascal, Visual Basic, and Java. Java in particular received much attention. More radical and innovative than the RAD languages were the new scripting languages. These did not directly descend from other languages and featured new syntaxes and more liberal incorporation of features. Many consider these scripting languages to be more productive than even the RAD languages, but often because of choices that make small programs simpler but large programs more difficult to write and maintain. Nevertheless, scripting languages came to be the most prominent ones used in connection with the Web.

Some important languages that were developed in this period include:

11.6 Current trends

Programming language evolution continues, in both industry and research. Some of the current trends include:

- Increasing support for functional programming in mainstream languages used commercially, including pure functional programming for making code easier to reason about and easier to parallelise (at both micro- and macro- levels)
- Constructs to support concurrent and distributed programming.
- Mechanisms for adding security and reliability verification to the language: extended static checking, dependent typing, information flow control, static thread safety.
- Alternative mechanisms for composability and modularity: mixins, traits, delegates, aspects.
- Component-oriented software development.
- Metaprogramming, reflection or access to the abstract syntax tree
- Increased emphasis on distribution and mobility.
- Integration with databases, including XML and relational databases.
- Support for Unicode so that source code (program text) is not restricted to those characters contained in the ASCII character set; allowing, for example, use of non-Latin-based scripts or extended punctuation.
- XML for graphical interface (XUL, XAML).
- Open source as a developmental philosophy for languages, including the GNU compiler collection and recent languages such as Python, Ruby, and Squeak.
- AOP or Aspect Oriented Programming allowing developers to code by places in code extended behaviors.
- Massively parallel languages for coding 2000 processor GPU graphics processing units and supercomputer arrays including OpenCL

Some important languages developed during this period include:

11.7 Prominent people

Some key people who helped develop programming languages (in alpha order):

- Joe Armstrong, creator of Erlang.
- John Backus, inventor of Fortran.
- Alan Cooper, developer of Visual Basic.
- Edsger W. Dijkstra, developed the framework for structured programming.
- Jean-Yves Girard, co-inventor of the polymorphic lambda calculus (System F).
- James Gosling, developer of Oak, the precursor of Java.
- Anders Hejlsberg, developer of Turbo Pascal, Delphi and C#.
- Rich Hickey, creator of Clojure.
- Grace Hopper, developer of Flow-Matic, influencing COBOL.
- Jean Ichbiah, chief designer of Ada, Ada 83
- Kenneth E. Iverson, developer of APL, and co-developer of J along with Roger Hui.
- Alan Kay, pioneering work on object-oriented programming, and originator of Smalltalk.
- Brian Kernighan, co-author of the first book on the C programming language with Dennis Ritchie, coauthor of the AWK and AMPL programming languages.
- Yukihiro Matsumoto, creator of Ruby.
- John McCarthy, inventor of LISP.
- Bertrand Meyer, inventor of Eiffel.
- Robin Milner, inventor of ML, and sharing credit for Hindley–Milner polymorphic type inference.
- John von Neumann, originator of the operating system concept.
- Martin Odersky, creator of Scala, and previously a contributor to the design of Java.
- John C. Reynolds, co-inventor of the polymorphic lambda calculus (System F).
- Dennis Ritchie, inventor of C. Unix Operating System, Plan 9 Operating System.
- Nathaniel Rochester, inventor of first assembler (IBM 701).
- Guido van Rossum, creator of Python.

- Bjarne Stroustrup, developer of C++.
- Ken Thompson, inventor of B, Go Programming Language, Inferno Programming Language, and Unix Operating System co-author.
- Larry Wall, creator of the Perl programming language (see Perl and Perl 6).
- Niklaus Wirth, inventor of Pascal, Modula and Oberon.
- Stephen Wolfram, creator of Mathematica.

11.8 See also

11.9 References

[1] J. Fuegi and J. Francis (October–December 2003), “Lovelace & Babbage and the creation of the 1843 'notes'", *Annals of the History of Computing* **25** (4): 16, 19, 25, doi:10.1109/MAHC.2003.1253887

[2] Rojas, Raúl, et al. (2000). “Plankalkül: The First High-Level Programming Language and its Implementation”. Institut für Informatik, Freie Universität Berlin, Technical Report B-3/2000. (full text)

[3] Sebesta, W.S Concepts of Programming languages. 2006;M6 14:18 pp.44. ISBN 0-321-33025-0

[4] Knuth, Donald E.; Pardo, Luis Trabb. “Early development of programming languages”. *Encyclopedia of Computer Science and Technology* (Marcel Dekker) **7**: 419–493.

[5] Peter J. Bentley (2012). *Digitized: The Science of Computers and how it Shapes Our World*. Oxford University Press. p. 87.

[6] Hopper (1978) p. 16.

[7] Sammet (1969) p. 316

[8] Sammet (1978) p. 204.

11.10 Further reading

- Rosen, Saul, (editor), *Programming Systems and Languages*, McGraw-Hill, 1967
- Sammet, Jean E., *Programming Languages: History and Fundamentals*, Prentice-Hall, 1969
- Sammet, Jean E. (July 1972). “Programming Languages: History and Future”. *Communications of the ACM* **15** (7): 601–610. doi:10.1145/361454.361485.
- Richard L. Wexelblat (ed.): *History of Programming Languages*, Academic Press 1981.
- Thomas J. Bergin and Richard G. Gibson (eds.): *History of Programming Languages*, Addison Wesley, 1996.

11.11 External links

- History and evolution of programming languages.
- Graph of programming language history

11.12 Text and image sources, contributors, and licenses

11.12.1 Text

Bacon1234321, TerryAlex and Anonymous: 219

- **Firmware** *Source:* http://en.wikipedia.org/wiki/Firmware?oldid=649489764 *Contributors:* Damian Yerrick, Zundark, The Anome, Tarquin, Andre Engels, Enchanter, Hannes Hirzel, Rbrwr, Nixdorf, Mac, Smack, Dysprosia, Myshkin, Furrykef, Wernher, Robbot, Zz, Techtonik, Bkell, Wikibot, Engerim, Alan Liefting, David Gerard, DavidCary, Kenny sh, Antandrus, Beland, Panit, Abdull, Adashiel, Vsmith, Jojit fb, Lysdexia, Alansohn, Guy Harris, Spangineer, Brock, Wtshymanski, TahitiB, Dzordzm, SCEhardt, Isnow, Graham87, Raffaele Megabyte, NeonMerlin, Fred Bradstadt, Titoxd, FlaBot, SchuminWeb, Ayla, Fresheneesz, BMF81, King of Hearts, Chobot, Nastajus, WriterHound, YurikBot, Borgx, MMuzammils, Stephenb, Manop, Sikon, Cunado19, Jabencarsey, Code65536, Bota47, Nlu, Gregzeng, Nikkimaria, GrinBot, Perardi, NetRolller 3D, SmackBot, YellowMonkey, Redslime, F, Wlindley, Hydrogen Iodide, Gribeco, Ultramandk, Evanreyes, Brianski, Jcarroll, Dlohcierekim's sock, Frap, Chlewbot, Steveo1544, LouScheffer, Joema, Decltype, DylanW, NickPenguin, Autopilot, Weatherman1126, AThing, Mr Stephen, UKER, Dicklyon, Bashari, Amitch, The7thmagus, CapitalR, Sph147, Tawkerbot2, Ryt, Nuclearo, Sandeep pranavam, HenkeB, SolarisBigot, Phatom87, O mores, MikeLacey, Normix, Stevag, Thijs!bot, Kubanczyk, Fourchette, Dgies, Icep, Mentifisto, Widefox, Nosbig, JAnDbot, Arch dude, Aki009, ProjectPlatinum, VoABot II, Kuyabribri, Tedickey, SomethingWittyHere, Alex Spade, Gwern, Majesty9012, Herbythyme, Public Menace, Thaurisil, Laurusnobilis, JensRex, LordAnubisBOT, Cometstyles, STBotD, DorganBot, Ertyeryery, Aaronsingh, CardinalDan, Idioma-bot, Deor, Philip Trueman, Rocketmagnet, TyrantX, WolfgangEcker, DennyColt, LeaveSleaves, Mazarin07, Softtest123, Miko3k, Alaniaris, SieBot, WereSpielChequers, Winchelsea, Stonejag, Steven Zhang, OKBot, Dillard421, ClueBot, Meekywiki, ChandlerMapBot, Posix memalign, SchreiberBike, Zappa711, Joel Saks, Darkicebot, S1fw, Rror, Cmr08, Jaymacdonald, Kurniasan, Dsimic, Addbot, Grandscribe, Cptnoremac, Colcolstyles, Fluffernutter, Cst17, Aclews56, ChenzwBot, 5 albert square, Tide rolls, Rainbow will, Another-anomaly, Crt, Luckas-bot, Yobot, Ptbotgourou, Fraggle81, Legobot II, Crispmuncher, MrBurns, AnomieBOT, Exp HP, Rubinbot, Name5555, Gacpro, Xqbot, Rijndael, The Evil IP address, GrouchoBot, RibotBOT, Universalss, Fobeteh, Erik9, SupportFTP, FrescoBot, Mohandesi, Winterst, SpaceFlight89, Jandalhandler, Siddharthsivakumar, TobeBot, SchreyP, RjwilmsiBot, Jtsandlund, Lopifalko, WikitanvirBot, Chewbaca75, Liquidmetalrob, Doomedtx, Contribute23, The contributor 4783, Chezi-Schlaff, MaGa, ChuispastonBot, VictorianMutant, Kenny Strawn, Rocketrod1960, ClueBot NG, Rajaram Sarangapani, Steve dexon, Wbm1058, BG19bot, Walk&check, Arashium, Mark Arsten, Compfreak7, Rancher 42, Winston Chuen-Shih Yang, BattyBot, Tkbx, Tagremover, Bachware, Ajv39, Marian Robinson, NoBearHere, Melonkelon, Gerardwm, Shaddycrook, Comp.arch, Monkbot, Guglastican, Tanankmaster118, MXocrossIIB, Garfield Garfield, Ggordonbyrne and Anonymous: 314

- **Computer multitasking** *Source:* http://en.wikipedia.org/wiki/Computer%20multitasking?oldid=649350501 *Contributors:* Damian Yerrick, Bryan Derksen, The Anome, Ap, Rjstott, Dachshund, Pit, Rp, Theanthrope, Ahoerstemeier, Salsa Shark, Magnus.de, Jessel, Tpbradbury, Wernher, Robbot, Tobias Bergemann, Jyril, Lupin, Peruvianllama, AJim, AlistairMcMillan, Nayuki, Edcolins, Wmahan, Tipiac, Starblue, Stephan Leclercq, Zhuuu, Beland, QuiTeVexat, Simson, Burschik, Ulmanor, Abdull, Mormegil, CALR, Rich Farmbrough, Florian Blaschke, Djce, Dmeranda, Bender235, Lou Crazy, CanisRufus, Drhex, Bobo192, R. S. Shaw, Csabo, Wtshymanski, H2g2bob, Mikenolte, Nuno Tavares, Gimboid13, Palica, Pabix, Guinness2702, Bhadani, GeorgeBills, FlaBot, Chris Purcell, RexNL, YurikBot, Wavelength, DMahalko, Piet Delport, Zimbricchio, CarlHewitt, Dianne Hackborn, Misza13, DeadEyeArrow, PS2pcGAMER, JakkoWesterbeke, Ketsuekigata, Drable, Thomas Blomberg, GrinBot, SmackBot, Reedy, Ixtli, Sparking Spirit, Andy M. Wang, Stevage, Nbarth, Rrelf, Tsca.bot, AntiVan, Zvar, Jsavit, Radagast83, FormerUser1, Esb, Ozhiker, Harryboyles, BlindWanderer, Loadmaster, Dicklyon, EdC, Peyre, Iridescent, Tawkerbot2, FatalError, Unixguy, Ale jrb, Phatom87, Xaariz, Kubanczyk, Wikid77, Marek69, AntiVandalBot, Ad88110, Alphachimpbot, JAnDbot, MER-C, Arch dude, Magioladitis, JNW, Faizhaider, Web-Crawling Stickler, Glen, Manticore, Maurice Carbonaro, Public Menace, Jesant13, Silas S. Brown, Cometstyles, Jcea, VolkovBot, Mazarin07, HiDrNick, Toyalla, SieBot, RucasHost, Theaveng, Yerpo, Ixe013, Nskillen, Frappucino, Anchor Link Bot, Mhouston, ClueBot, Ndenison, M4gnum0n, PixelBot, Estirabot, Sun Creator, Oliverbell99, Fiskegalen92, Andy16666, DumZiBoT, Joel Saks, Kwjbot, Dsimic, Arkantospurple, Deineka, Akshatdabralit, Cst17, CarsracBot, Lightbot, דוד שי, Gail, Zorrobot, John.St, Legobot, Luckas-bot, ArchonMagnus, Peter Flass, AnomieBOT, TwistedText, Materialscientist, GB fan, Xqbot, Julle, Almabot, 399man, RibotBOT, Alan.A.Mick, Rat2, D'ohBot, Mfwitten, Atlantia, EmausBot, WikitanvirBot, Dewritech, Tolly4bolly, L Kensington, ChuispastonBot, 28bot, ClueBot NG, Scicluna93, Toastyking, Mesoderm, ScottSteiner, Theopolisme, Compilation finished successfully, Jimperio, GGShinobi, Venera Seyranyan, ANI MARTIROSYAN, Shaun, BattyBot, FizzixNerd, Wrmattison, Mogism, Jamesx12345, Sriharsh1234, Vcfahrenbruck, OMPIRE, Benjamintf1, 7stone7, AyanMazumdar91 and Anonymous: 191

- **Time-sharing** *Source:* http://en.wikipedia.org/wiki/Time-sharing?oldid=647367226 *Contributors:* Derek Ross, Dachshund, Maury Markowitz, Stevertigo, Michael Hardy, Geoffrey, Docu, EdH, Magnus.de, Fuzheado, Greenrd, Lfwlfw, Saltine, Shizhao, Robbot, Nick Pisarro, Jr., Tobias Bergemann, Pretzelpaws, Macrakis, VampWillow, Andycjp, Phe, Deh, Dyl, Cedders, R. S. Shaw, Cohesion, Guy Harris, Diego Moya, M7, RainbowOfLight, SteinbDJ, Roland2, Kelly Martin, DonPMitchell, Pmcjones, Graham87, FreplySpang, Predius, WCFrancis, Toresbe, Eubot, GreyCat, Chobot, YurikBot, DMahalko, Msikma, Robertvan1, Rwwww, GrinBot, Armin76, KnightRider, SmackBot, Skizzik, Snori, Elagatis, LaggedOnUser, Alinefr, Nbarth, Wootmaster, Zalmoxe, Lguzenda, Tlesher, IronGargoyle, 16@r, Loadmaster, Wws, NormHardy, Quibik, Thijs!bot, Alphachimpbot, Steveprutz, SteveSims, Antipodean Contributor, GermanX, Gwern, Al Kossow, Gregory haynes, Mindgames11, Trusilver, Huey45, Wa3frp, Public Menace, Dispenser, Uhai, DorganBot, VolkovBot, Anajemstaht, TedColes, Patelski, Sources said, BloodDoll, WRK, Herbnet, Rvonder, MenoBot, Bbump, DumZiBoT, Baudway, Jabberwoch, Ghettoblaster, MrOllie, Lightbot, Legobot, Luckas-bot, Yobot, Peter Flass, AnomieBOT, Pragmatiste, Materialscientist, XZeroBot, Prunesqualer, FrescoBot, Cbonnert, W Nowicki, Winterst, Turtlecom, Skyerise, Mattrod666, Kmnamee, ZéroBot, Lilianag, W163, Donner60, ClueBot NG, Blitzmut, Snotbot, Kangaroopower, Creichardt0130, Numbermaniac, Jamesx12345, Huihermit and Anonymous: 103

- **Real-time computing** *Source:* http://en.wikipedia.org/wiki/Real-time%20computing?oldid=645623988 *Contributors:* Damian Yerrick, The Anome, WillWare, Alex, Khendon, Aldie, Stevertigo, Frecklefoot, Patrick, Mac, BigFatBuddha, Tristanb, Kaihsu, Greenrd, Furrykef, Wernher, Bloodshedder, Pakaran, Wikibot, Giftlite, DavidCary, Mintleaf, BenFrantzDale, Levin, Sietse, Tweenk, Neilc, Electrawn, Sam Hocevar, Bluefoxicy, Abdull, RevRagnarok, Ulflarsen, Fuffzsch, Discospinster, Smyth, S.K., RoyBoy, Quaternion, Krischik, Suruena, P garyali, Nuno Tavares, Ruud Koot, The Nameless, Graham87, JoelG, Ronnotel, Jerickson314, Cjoev, GeorgeBills, Eubot, Arnero, EamonnPKeane, Roboto de Ajvol, YurikBot, Borgx, Ori.livneh, Gsathish, JulesH, ScottyWZ, TERdON, 2over0, CIreland, That Guy, From That Show!, SmackBot, Redslime, Ianb1469, Janm67, Tripledot, Frap, Cybercobra, MParaz, Jwy, MureninC, Tompsci, Fitzhugh, Gvf, Mdsharpe, D o m e, Kvng, Lee Carre, Woodroar, Wleizero, Vocaro, FatalError, CmdrObot, Ultimus, FabgrOOv, JMatthews, Al Lemos, WhaleyTim, Afabbro, AntiVandalBot, Cbs228, Ste4k, Lperez2029, Bongwarrior, Britton ohl, Pitagora, Gwern, Mange01, ProfessorDJF, Dvanaken, Public Menace, Lonesomefighter, Treisijs, Dindon, Simonjwright, Jozue, Sibin m, Jackfork, Bearian, Softtest123, Codeispoetry, NHSKR, ClueBot, The Thing That Should Not Be, Wolfch, Benabomb1, Niemeyerstein en, Niceguyedc, Teutonic Tamer, Semitransgenic, XLinkBot, FellGleaming, Dekart, Dsimic, Addbot, Ghettoblaster, Mathew Rammer, GyroMagician, Graham.Fountain, Flamminifra, Yobot, TaBOT-zerem, Peter Flass, Cosmiclazer, AnomieBOT, JJ-Stern, Groovenstein, Omnipaedista, MightyMaven, Shadowjams, A.amitkumar, FrescoBot, StaticVision, Phanhaitrieu, Alexeicolin, Ecoman24, RenamedUser01302013, A930913, L Kensington, Rmashhadi, Rocketrod1960, Planetscared, ClueBot NG, Andrew McRae 78, Matthiaspaul, Massimiliano Carli, Neotheone1981, Danim, Helpful Pixie Bot, BG19bot, Walk&check, Magdjtk, Mohmdbilal, Yelojakit, Anjublaxs, Kodign, Harlem Baker Hughes, Mc2labs, Amenychtas and Anonymous: 194

- **Fault tolerance** *Source:* http://en.wikipedia.org/wiki/Fault%20tolerance?oldid=650112739 *Contributors:* DavidCary, Rchandra, Beland, Neutrality, Conan, BMF81, StuRat, Kvng, Adrian J. Hunter, Alikaalex, SimonTrew, Pianist ru, Fraggle81, AnomieBOT, BG19bot, MusikAnimal, Closeddoors1559, MaryEFreeman, Amenychtas, Monkbot, Andrewstone and Anonymous: 4
- **Mean time between failures** *Source:* http://en.wikipedia.org/wiki/Mean%20time%20between%20failures?oldid=637628726 *Contributors:* SimonP, Youandme, JohnOwens, Michael Hardy, Wapcaplet, Ixfd64, Tomi, Delirium, Ronz, Den fjättrade ankan, Nnh, Owen, Robbot, Chealer, Panthouse, Bkell, Mat-C, Emrys2, Alanl, Beiss, Art LaPella, Causa sui, Guy Harris, Atlant, ShunterAlhena, Wyatts, Versageek, Koavf, Apapadop, Aspro, Chobot, Fabartus, Gerben49, Kyle Barbour, Habbie, SmackBot, AnOddName, Ppntori, BrynJones216, Bluebot, Thumperward, MalafayaBot, Nbarth, Aaron Lawrence, Arkrishna, JoeBot, CmdrObot, David Santos, Judygs, BetacommandBot, Pogogunner, AntiVandalBot, Carewolf, JAnDbot, MER-C, Flippin42, Brownout, Dtroncho, Rocinante9x, R'n'B, Wiki Raja, EdBever, Bin95, Anton Khorev, Skullers, Rubyuser, Sdsds, Melsaran, Seb az86556, Tomaxer, EgbertE, Root Beers, AlleborgoBot, Finnrind, Sunookitsune, Tomalak geretkal, NYCDA, Johnqtodd, Elnon, Bassplr19, PipepBot, PetterEkhem, Ashchori, Andjohn2000, Ferdna, Ppejack, Cincaipatrin, Boulty, Scuarty, Qwfp, Maurizio.Cattaneo, Dsimic, Scasa155, Tayste, Addbot, Lookskywatcher, LatitudeBot, Lunchsure, SpBot, Ettrig, Luckas-bot, Yobot, AnomieBOT, Piano non troppo, Materialscientist, Xqbot, RibotBOT, SassoBot, Elavierijr, Alexdfromald, Jodypro, EmausBot, Ibbn, Telcoterry, Empty Buffer, Gapeev, خالقیان, ClueBot NG, AeroPsico, Helpful Pixie Bot, Linear77, Weakestletter, Honza889, Zalatos, Jpetitbon and Anonymous: 134
- **Flowchart** *Source:* http://en.wikipedia.org/wiki/Flowchart?oldid=650120814 *Contributors:* Damian Yerrick, Stevertigo, Michael Hardy, Booyabazooka, Norm, Rp, Wapcaplet, GTBacchus, Haakon, Ronz, Darkwind, Poor Yorick, Frieda, Tacvek, Jm34harvey, Sbwoodside, Bevo, Robbot, PBS, Romanm, Phatsphere, Anthony, Tobias Bergemann, Giftlite, Andromeda, Kim Bruning, Mintleaf, Tom harrison, Bensaccount, Tusharvjoshi, Apsio, Edcolins, Geni, Antandrus, Xinit, Tels, BoP, Dcandeto, Andreas Kaufmann, Rob cowie, Mormegil, Discospinster, Luvcraft, Pixel8, LeeHunter, ZeroOne, Nabla, Bobo192, WikiLeon, Minghong, SeanGustafson, Mdd, Patsw, Gary, Pinar, Diego Moya, Zeborah, Mysdaao, Jost Riedel, Wtshymanski, Hawky, ScotS, Kazvorpal, Forteblast, Bobrayner, Nuno Tavares, Goodgerster, Trapolator, Triddle, Davidfstr, Roundand, Isnow, Mandarax, Sin-man, Knudvaneeden, Kane5187, Crazynas, Ligulem, BasBloemsaat, The wub, Ttwaring, Jobarts, Gurch, Czar, Threner, Sstrader, Vonkje, DVdm, Gdrbot, YurikBot, Wavelength, Borgx, Jzylstra, TriniTriggs, RussBot, Michael Slone, Bergsten, Akamad, Rsrikanth05, NawlinWiki, Welsh, Jon1012, Aaron Brenneman, Dethomas, Mjchonoles, RUL3R, DeadEyeArrow, Ejl, Jadster, Nacimota, Closedmouth, Nae'blis, LeonardoRob0t, Gesslein, Cmglee, AndrewWTaylor, EJSawyer, Hpmagic, SmackBot, Haza-w, Jfgrcar, Chelmite, Alksub, Jhedemann, Frymaster, BiT, Gilliam, Toddintr, Ohnoitsjamie, Timbouctou, Jjalexand, Persian Poet Gal, Lazyquasar, MK8, Emufarmers, Silly rabbit, SchfiftyThree, Metacomet, Aoreias, Octahedron80, Kostmo, Chendy, Xchbla423, DanielJudeCook, JonHarder, Juandev, Rrburke, Seattlenow, Alx xlA, Demoeconomist, Dreadstar, Doodle77, Pmarc, FelisLeo, Kukini, Rklawton, Kuru, Ckatz, Noah Salzman, Wikidrone, Yaxh, Squirepants101, Hu12, Mkoyle, Levineps, Fireman.sparkey, Iridescent, Dreftymac, Pimlottc, Blehfu, Zfang, Tfinneid, Tawkerbot2, George100, Amniarix, GeordieMcBain, JForget, DevinCook, Slazenger, Ajikoe, Gogo Dodo, Kallerdis, ThreeVryl, Csodessa, Ebyabe, Bayonetblaha, Thijs!bot, Epbr123, Daa89563, Headbomb, Marek69, A3RO, Void Ptr, Leon7, Escarbot, Mentifisto, AntiVandalBot, Seaphoto, VectorPosse, FirefoxRocks, Isilanes, Dylan Lake, SoftwareSalesRep, LéonTheCleaner, Res2216firestar, Andrzejbanas, Deadbeef, JAnDbot, Ndyguy, ThomasO1989, MER-C, PhilKnight, Fourchannel, Aki009, Raanoo, Dixius99, VoABot II, Swpb, Skew-t, Irrelative, Indon, Abednigo, Minion o' Bill, Martynas Patasius, DerHexer, Philg88, Hbent, Cocytus, Gwern, MartinBot, TheInfluence, JBakaka, CommonsDelinker, Steve5682, Erkan Yilmaz, J.delanoy, Trusilver, Rgoodermote, Numbo3, Kaizer1784, Ginsengbomb, Eliz81, SCB '92, Bhanks, Iain marcuson, Schaaftin, Pmiller42au, Dpdearing, Edrawing, Skier Dude, Davandron, SuzanneKn, NewEnglandYankee, Cometstyles, Jamesontai, Taffykins, Tbone762, Informavores, Bonadea, Chsimps, Shiva.rock, SoCalSuperEagle, VolkovBot, Cassovian, Jeff G., WOSlinker, YewBowman, Zidonuke, Planetary Chaos, Chartex, JayC, Jimrogerz, Sotruetwo, Gramware, Slash454, DaisyN, Kuppuz, UnitedStatesian, Sychen, Jamelan, Economist332, Wortech tom, Madhero88, Zhenqinli, Priyanshu hbti, Falcon8765, Eye of slink, Insanity Incarnate, Dusti, Flyer22, Qst, Dwiakigle, Pm master, EnOreg, Oxymoron83, Jdaloner, Hobartimus, Sunrise, Zaq 42, Mygerardromance, Haris.tv, Denisarona, Nishadha, Martarius, ClueBot, Ohedland, Rjd0060, Jan1nad, Bjornwireen, Ancos, Lampak, Gordon Ecker, NathanWalther, SpikeToronto, Lartoven, Rhododendrites, DeltaQuad, ChrisHodgesUK, Johnuniq, SoxBot III, Bücherwürmlein, DumZiBoT, XLinkBot, Cycn, Libcub, Skarebo, WikHead, NellieBly, Tcreg010, Maimai009, Addbot, ERK, Proofreader77, Kongr43gpen, MrOllie, Jimmcgovern15, Numbo3-bot, LarryJeff, Tide rolls, Sbasan, Lightbot, Avono, JakobVoss, Juzzierules97, Luckas-bot, Yobot, Cflm001, Mauler90, GateKeeper, Agent phoenex, AnomieBOT, Rubinbot, Jim1138, IRP, Piano non troppo, 90, AdjustShift, Materialscientist, Citation bot, ArthurBot, Ayda D, Xqbot, Spidern, The sock that should not be, Capricorn42, 4twenty42o, Jmundo, Tomwsulcer, J04n, Stratocracy, Darrellswain, GainLine, Shadowjams, Davidobrisbane, FrescoBot, Canton Viaduct, Inc ru, Artem M. Pelenitsyn, Kaziali, Green Tea Writer, Techturtle, Beniganj, Supercuban, RockfangSemi, Pinethicket, HRoestBot, Edderso, RedBot, Pweemz, SpaceFlight89, Bgpaulus, Viswanath1947, DC, Gryllida, TobeBot, Mikhailcazi, Vrenator, Suffusion of Yellow, Tbhotch, RobertMfromLI, LoStrangolatore, Blue.eyed.girly, DASHBot, EmausBot, Gfoley4, Sumsum2010, Georgie30.11.96, Jsvforever, Tommy2010, Wikipelli, K6ka, ZéroBot, John Cline, Iwillgoogleitforyou, ZweiOhren, Makecat, DiamFC, OnePt618, Tolly4bolly, Thine Antique Pen, HupHollandHup, Donner60, 28bot, Cgt, ClueBot NG, Keith caly, Satellizer, Movses-bot, RobotEducativo, Widr, Eucolopo, Shanmanatpoi, Oddbodz, Strike Eagle, DBigXray, BG19bot, Pine, Vagobot, Владимир Паронджанов, AvocatoBot, Paoloss, Altaïr, LongLiveMusic, Jurjenz04, Klilidiplomus, Pito Pup, Mike Kue, Cimorcus, RichardMills65, StarryGrandma, AuthorizeditorA, Electricmuffin11, EuroCarGT, JYBot, Kushalbiswas777, Cwobeel, Maryeputnam, GigaGerard, I am One of Many, FrigidNinja, Joelee99, Konyservant, Ugog Nizdast, Ginsuloft, Monkbot, AKS.9955, IvanZhilin, U2fanboi, Theo rosicky, Kamran.Rokni, Surell24, Miasusu and Anonymous: 686
- **Programming language** *Source:* http://en.wikipedia.org/wiki/Programming%20language?oldid=650001603 *Contributors:* Magnus Manske, Matthew Woodcraft, Derek Ross, LC, Brion VIBBER, Mav, Koyaanis Qatsi, AstroNomer, Jeronimo, Ap, Malcolm Farmer, Alex, Rjstott, Andre Engels, Fubar Obfusco, SimonP, Merphant, FvdP, Imran, Rlee0001, B4hand, Stevertigo, Hfastedge, DennisDaniels, Edward, K.lee, Michael Hardy, Tim Starling, Booyabazooka, Kwertii, Nixdorf, MartinHarper, Ixfd64, TakuyaMurata, Karingo, Minesweeper, Ahoerstemeier, Nanshu, Angela, Kragen, Poor Yorick, Nikai, Andres, Grin, Evercat, TonyClarke, ^{212}, Jonik, Mxn, Vivin, Speuler, Dave Bell, Dcoetzee, Reddi, Ww, Mac c, Dysprosia, Jitse Niesen, Gutza, Doradus, Zoicon5, 2988, Taxman, ZeWrestler, Bevo, Jph, Jusjih, David.Monniaux, Mrjeff, Finlay McWalter, Pumpie, AlexPlank, Robbot, Noldoaran, Murray Langton, Friedo, Fredrik, Thniels, RedWolf, Altenmann, SmartBee, Romanm, Rursus, Wlievens, Hadal, Borislav, Lupo, BigSmoke, Gwicke, Tobias Bergemann, Ancheta Wis, Gploc, Centrx, Giftlite, Thv, Dtaylor1984, Akadruid, PaulFord, Cobaltbluetony, Everyking, Esap, Wikibob, Mellum, Jorend, AJim, Bonaovox, Behnam, Ptk, Macrakis, VampWillow, Bobblewik, Wmahan, Vadmium, Quagmire, Yath, Beland, Elembis, Jossi, Phil Sandifer, DenisMoskowitz, Marcos, RainerBlome, Addicted2Sanity, Joyous!, Positron, Quota, Teval, Zondor, Damieng, EagleOne, Gazpacho, Mike Rosoft, Brianjd, SimonEast, Yana209, Noisy, Zaheen, Rich Farmbrough, Leibniz, Jpk, HeikoEvermann, Lulu of the Lotus-Eaters, LindsayH, Michael Zimmermann, Paul August, ESkog, Kbh3rd, Ben Standeven, Danakil, CanisRufus, Hayabusa future, Shanes,

Bobo192, AmosWolfe, Mpils, L33tminion, SpeedyGonsales, Photonique, Tgr, Saccade, Kickstart70, Sam Korn, Phyzome, Simplyanil, Alansohn, Liao, Guy Harris, Diego Moya, Jeltz, Krischik, Suruena, Omphaloscope, Zawersh, Kinema, HenryLi, KTC, Forderud, Oleg Alexandrov, Mahanga, Revived, Infinoid, Roland2, Hoziron, Woohookitty, NewbieDoo, Nuggetboy, Ilario, Oldadamml, Nameneko, Ruud Koot, Goodgerster, Brentdax, KymFarnik, Dolfrog, Davidfstr, Mangojuice, Toussaint, Marudubshinki, Yoric, Chun-hian, Kbdank71, Mendaliv, Pwv1, Icey, Ketiltrout, Sjakkalle, Rjwilmsi, Tizio, Angusmclellan, Koavf, Swirsky, Wikibofh, Amire80, Quiddity, Oblivious, Ligulem, DouglasGreen, Bubba73, Reinis, GregAsche, JanSuchy, Fantom, FlaBot, Mathbot, Undeference, Nihiltres, Harmil, Vsion, Xavier Combelle, RexNL, Ewlyahoocom, Gurch, Mpradeep, Quuxplusone, Tysto, Windharp, Chobot, Bgwhite, ColdFeet, Wavelength, Hairy Dude, RussBot, Hyad, Robert A West, Taejo, Davidpdx, Bhny, Piet Delport, KSmrq, SpuriousQ, Edward301, Stephenb, Gaius Cornelius, Rsrikanth05, Wimt, CarlHewitt, EngineerScotty, NawlinWiki, Wiki alf, Jaxl, Johann Wolfgang, BirgitteSB, Mccready, Brandon, Jpbowen, JulesH, Sekelsenmat, Nick C, Tony1, Slaad, Natkeeran, BOT-Superzerocool, DeadEyeArrow, Perry Middlemiss, Ms2ger, Pooryorick, WAS 4.250, Novasource, Zero1328, Rushyo, Closedmouth, Jwissick, Kuciwalker, Cedar101, Peter, Donhalcon, Vahid83, Katieh5584, Kungfuadam, TuukkaH, DVD R W, SmackBot, Tarret, KnowledgeOfSelf, Lagalag, Vald, Brick Thrower, Alksub, Monz, ElAmericano, Xaosflux, Gilliam, Duke Ganote, Ohnoitsjamie, Bh3u4m, Bluebot, Klasbricks, LinguistAtLarge, JMSwtlk, Persian Poet Gal, SeeAnd, MK8, Jprg1966, Thumperward, Fplay, EncMstr, Victorgrigas, RayAYang, Nbarth, Royboycrashfan, Can't sleep, clown will eat me, Rrburke, Allan McInnes, -Barry-, Cybercobra, Nick125, MisterCharlie, HarisM, Hammer1980, DMacks, MegaHasher, Vriullop, Derek farn, Waterfles, Zarniwoot, Antonielly, JohnWittle, Ckatz, 16@r, A. Parrot, Eivind F Øyangen, Dan128, Slakr, Alhoori, Pieguy48, Xionbox, Dl2000, SubSeven, Hu12, Stephen B Streater, Iridescent, Dreftymac, DavidHOzAu, Tawkerbot2, Vkhaitan, Switchercat, INkubusse, Acacix, Ahy1, Georg Peter, Kris Schnee, Green caterpillar, Burkedavis, Jaxad0127, ShelfSkewed, MarsRover, Napi, Ezrakilty, Charlie Huggard, Ubiq, Krauss, RenamedUser2, Fl, Peterdjones, Jason5ayers, Blaisorblade, Torc2, NotQuiteEXPComplete, Mattisse, Malleus Fatuorum, Epbr123, Qwyrxian, N5iln, Headbomb, John254, Merbabu, Ideogram, AccurateOne, Natalie Erin, Escarbot, Acaciz, AntiVandalBot, Tewy, Gioto, Luna Santin, Seaphoto, Wildboy211, Prolog, Jj137, Science History, VictorAnyakin, KHaskell, JAnDbot, JaK81600, MER-C, IanOsgood, Andonic, Hut 8.5, PhilKnight, Four Dog Night, VoABot II, Necklace, Foobah, Tedickey, Catgut, Indon, JohnLai, Abednigo, Mkdw, ArmadilloFromHell, DerHexer, Philg88, Khalid Mahmood, Calltech, Gwern, Kiminatheguardian, MartinBot, Benjaminct, Mschel, Autocratique, Tgeairn, J.delanoy, Pharaoh of the Wizards, Kimse, Trusilver, Giorgios (usurped), Ntalamai, Abeliavsky, Macaldo, Jesant13, It Is Me Here, LordAnubisBOT, McSly, Dominator09, SparsityProblem, Raise exception, NewEnglandYankee, Ohms law, Cmichael, Ultra two, Juliancolton, LordCo Centre, Don't Copy That Floppy, Useight, Steel1943, Idioma-bot, Funandtrvl, Reelrt, Ottershrew, Red Thrush’s Bot, VolkovBot, Cireshoe, Philip Trueman, Muro de Aguas, Charlesriver, Tomatensaft, Lradrama, K12308025, DragonLord, Sgbirch, Noformation, MearsMan, PlayStation 69, Yk Yk Yk, Cnilep, AlleborgoBot, Knyf, S.Örvarr.S, EJF, SieBot, Sonicology, Tiddly Tom, Speed Air Man, Bill122, Krawi, Timhowardriley, Logarkh, Jerryobject, Mwaisberg, Happysailor, Flyer22, Ranafon, Faradayplank, Lightmouse, Techman224, BenoniBot, Shane A. Bender, Chillum, ClueBot, CSProfBill, The Thing That Should Not Be, VsBot, Alliswellthen, Kedearian, Mild Bill Hiccup, Blanchardb, Robert Skyhawk, Iwantitallllllllll, Skytreader, Jotterbot, Hans Adler, Noosentaal, MelonBot, Chinabuffalo, SoxBot III, HopeChrist, Darkicebot, Naderi 8189, Cp15, Slashem, Noctibus, MystBot, Dsimic, Addbot, JPINFV, L Gottschalk, DOI bot, Ronhjones, Fieldday-sunday, Mentisock, Roux, LinkFA-Bot, Systemetsys, Wikisedia, Tide rolls, Avono, Gail, Jarble, Andylmurphy, Ben Ben, Legobot, Luckas-bot, Yobot, Midinastasurazz, OrgasGirl, Fraggle81, TaBOT-zerem, Abram.carolan, Pcap, Ningauble, Peter Flass, Conor123777, AnomieBOT, Nottsadol, 1exec1, Jim1138, Galoubet, Ulric1313, Lucian1900, Materialscientist, Citation bot, Maxis ftw, Fayt82, MauritsBot, Xqbot, Capricorn42, Jeffrey Mall, XZeroBot, Miym, GrouchoBot, Papercutbiology, RibotBOT, Gbruin, Russell Joseph McCann, Shadowjams, A.amitkumar, Apwestern, Captain-n00dle, Manpreett, Edgars2007, FrescoBot, Mark Renier, Mìthrandir, DivineAlpha, Citation bot 1, I dream of horses, Elockid, HRoestBot, 10metreh, Jonesey95, Jschnur, Σ, Robo Cop, Txt.file, Cmdodanli, ئاراس نوری, TheTechFan, Specs112, Diannaa, WikiTome, Suffusion of Yellow, Stroppolo, Vaibhavkanwal, DARTH SIDIOUS 2, Mean as custard, TomT0m, John lindgren, EmausBot, Orphan Wiki, Nima1024, Nutsnbolts222, Cogiati, Iuliatoyo, H3llBot, Karthikndr, Swatiri, Jguy, Noodleki, Carmichael, BioPupil, DASHBotAV, ClueBot NG, A520, Mesoderm, Asukite, Snickel11, Danim, Helpful Pixie Bot, DBigXray, Krenair, Kangaroopower, Wiki13, MusikAnimal, J991, Compfreak7, Loriendrew, Nbrothers, Thomasuniko, Usman&muzammal, Teammm, Bobbygammill, Amitkumargarg88, Dexbot, Rezonansowy, Majilis, Damian.rouson, Steamerandy, Kooginup, Seanhalle, Vanamonde93, Thetimperson, Grouphardev, Carrot Lord, François Robere, Wiki4Blog, Olmerta, Tentinator, Captain Conundrum, Komarov om, Alhade, Ginsuloft, Inaaaa, Danielmask, Lucky7-phool, Fastdrummer, Monkbot, Theanimalover, Abdallasyam, Wildkrat52, Whikie, Malistomailie, TheMacroChip, Papapasan, Bridgetlane, Lee Ann Dickerson and Anonymous: 727

- **History of programming languages** *Source:* http://en.wikipedia.org/wiki/History%20of%20programming%20languages?oldid=650213951 *Contributors:* Sabre23t, Rp, Fuzheado, Greenrd, Zoicon5, Phoebe, RedWolf, Altenmann, Rursus, Ancheta Wis, Pgan002, Duncandewar, Beland, Marc Mongenet, Ukexpat, Yuriz, Discospinster, Huffers, Lulu of the Lotus-Eaters, Ben Standeven, Danakil, Reeve, Szyslak, Sietse Snel, Smalljim, John Vandenberg, R. S. Shaw, Mdd, Jumbuck, Tablizer, Gerweck, Diego Moya, Sligocki, Mr700, Jost Riedel, Tony Sidaway, Zawersh, Djsasso, Mahanga, Firsfron, Ruud Koot, Wikiklrsc, Marudubshinki, Icey, Schmettow, Cassowary, Sceptre, Hairy Dude, Piet Delport, Rsrikanth05, Jpbowen, Moe Epsilon, GraemeL, JLaTondre, Banus, Rwwww, SmackBot, Skizzik, Chris the speller, Bluebot, UdayanBanerjee, Rheostatik, Ashawley, KevM, Pwjb, Lambiam, Lakinekaki, DiarmuidPigott, Aljullu, Kpengboy, JMK, DEddy, Pi, Pgr94, Cydebot, Valodzka, Indeterminate, Torc2, Juhovuori, Michael Fourman, Thijs!bot, Wikid77, Mojo Hand, Acaciz, Dylan Lake, Arch dude, IanOsgood, Antic-Hay, PhilKnight, Abcarter, VoABot II, B3tamike, Sammi84, Gwern, STBot, Macaldo, Entropy, Funandtrvl, VolkovBot, TXiKiBoT, T-bonham, Clarince63, Arohanui, Cnilep, Nibios, Monty845, Logan, SieBot, Malcolmxl5, Jrm2007, Happysailor, Oxymoron83, KoshVorlon, CharlesGillingham, The Thing That Should Not Be, Skäpperöd, DragonBot, Excirial, Hotcrocodile, MystBot, Totlmstr, Id1337x, Quinntaylor, Addbot, Mortense, Ghettoblaster, Chamal N, Griffin700, AnomieBOT, 1exec1, Jim1138, Materialscientist, Citation bot, Capricorn42, DSisyphBot, GrouchoBot, RibotBOT, FrescoBot, Alexanderaltman, Soxey6, NumberByColors, December21st2012Freak, Andrzejsliwa, Sinbadbuddha, Raees Iqbal, Gf uip, EmausBot, Bgeron, Tommy2010, Wikipelli, Cleme263, Vb4ever, ZéroBot, Tolly4bolly, Noodleki, Donner60, Denbosch, ClueBot NG, Jack Greenmaven, Hossein bardareh, Skolastika, ISTB351, PhnomPencil, Pankaj.nith, Kooku10, ChrisGualtieri, Christophe.billiottet, Khazar2, RaLusch, Jasssonpet, Nvourvachis, Epicgenius, I am One of Many, Jrgdelrisco, Tentinator, Paulo torrens, Babitaarora, Ugog Nizdast, Simmonsandrew75, Monkbot, Axiarchist, Anonymouscherrycake and Anonymous: 191

11.12.2 Images

- **File:8bit-dynamiclist.gif** *Source:* http://upload.wikimedia.org/wikipedia/commons/1/1d/8bit-dynamiclist.gif *License:* CC-BY-SA-3.0 *Contributors:* Own work *Original artist:* Seahen

- **File:Ambox_important.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/b/b4/Ambox_important.svg *License:* Public domain *Contributors:* Own work, based off of Image:Ambox scales.svg *Original artist:* Dsmurat (talk · contribs)

11.12.3 Content license