UNIVERSAL RESOURCE LOCATOR

In the World Wide Web global network information system (1), the names used to locate resources are called uniform resource locators (URL) (2). A key requirement of a wide-area network information system is the ability to name the location of resources in the system. Resources, such as documents and images, are distributed at various locations throughout the network. To access these resources, users of an information system must be able to name their locations.

URLs in the Web are similar in purpose to filenames used to locate files on a computer. Just as filenames enable people and programs to identify and refer to files on a computer, URLs enable people and programs to identify and refer to resources throughout the Web. And just as the components of a filename specify how to locate a file on a computer, the components of a URL specify how to locate a resource in the Web.

This article describes the syntax and semantics of URLs and how they are used to locate resources in the World Wide Web. The article first discusses the basic concepts of the Web as a background for discussing URLs. Then it describes the syntax and semantics of URLs and the use of relative URLs, in detail. The article then concludes with a discussion of related naming schemes for the Web.

BACKGROUND

Because uniform resource locators are intimately interwoven into the fabric of the World Wide Web, understanding the basic concepts of the Web helps in understanding the use and syntax of URLs. The Web is an information system that enables users to access resources across a global network. Resources are anything from documents, images, video streams, weather forecasts, and stock quotes to programs, such as Java applets (3). Users access and manipulate resources using a Web client, which is typically a Web browser such as Netscape Navigator (4) or Microsoft's Internet Explorer (5).

Resources are provided by services that run on servers. For example, the File Transport Protocol (FTP) (6) service provides clients access to files, and the HyperText Transport Protocol (HTTP) (7) service provides clients access to hypertext and multimedia documents, such as HyperText Markup Language (HTML) (8) documents. Other services include electronic mail, Usenet news, Gopher, and so on. Servers are the network hosts on which the services run, and they often run more than one service.

In a typical scenario for accessing a resource in the Web, a user first gives the URL for the resource to a Web client. The user specifies the URL in many different ways: by typing it in, selecting it from a set of bookmarks, pasting it from another document, or clicking on a URL embedded in a document. Because the URL names the location of the resource, the client then parses the URL to determine the server on which the resource is located and the service on the server that provides the resource. Then the client communicates with the service to access the resource, which typically involves downloading it into the Web client and displaying it to the user.

For example, to access the HTML document "http://www.w3.org/TheProject.html" with a Web client, a user gives the URL for the document to the client. The client then parses the URL to determine the server that has the document and communicates with the HTTP service on that server to locate the document. Then the HTTP service sends the document back to the client, and the client formats the document and displays it to the user in a window.

This example is a high-level overview of how URLs are used to locate resources in the Web. The next section details exactly how URLs are specified and how each URL component is parsed and used to determine resource location.

SYNTAX AND SEMANTICS

Resources in the World Wide Web can be accessed with many different services, such as HTTP, FTP, and electronic mail.

Table 1. Examples of URLs for the most common services in the World Wide Web. The second URL for a service, if any, is a more explicit version of the first URL.

Service	Example URL
Email	mailto:webmaster@w3.org
File	file://localhost/etc/motd
FTP	ftp://ds.internic.net/rfc/rfc959.txt
	ftp://anonymous:user@ds.internic.net:21/rfc/rfc959.txt
Gopher	gopher://boombox.micro.umn.edu:70/hh/gopher
HTTP	http://www.w3.org/Addressing/URL/Overview.html
	http://www.w3.org:80/Addressing/URL/Overview.html
Telnet	telnet:remote.host
Usenet News	news:comp.infosystems.www

For example, Table 1 shows a number of URLs that use the most common services found on the Web. And just as there are many different services of resources, there are many different schemes for describing the location of those resources. As a result, the general URL syntax is simple and generic, so that it encompasses all schemes for accessing resources. The URL begins with the name of the scheme, followed by a colon, followed by a scheme-specific locator:

⟨scheme⟩:⟨scheme-specific-locator⟩

For example, the electronic mail URL "mailto:webmaster@w3.org" uses the scheme "mailto" with the scheme-specific locator "webmaster@w3.org"" (an electronic mail address). Similarly, the HTTP URL "http://www.w3.org/Addressing" uses the scheme "http" with the scheme-specific locator "//www.w3.org/Addressing" (an HTML document).

The scheme-specific locator depends entirely on the scheme used to access the resource. In this way, the URL syntax is extensible. In the future, the Web will likely use new schemes to access both new and old types of resources, and the location of those resources using these schemes can still be specified with the URL syntax without having to change the syntax of existing schemes.

Because the syntax and semantics of the scheme-specific locator depends on the scheme, they can vary widely from one scheme to another. However, many schemes share a common syntax of five components:

⟨scheme⟩://⟨site⟩⟨path⟩?⟨query⟩#⟨fragment⟩

Each of these components is described later.

Scheme

The scheme component determines how to access the resource and how to parse the scheme-specific locator. The scheme name begins the URL and ends at the first colon. Scheme names are often the names of network services such as HTTP (http), FTP (ftp), Gopher (gopher), Usenet News (news), and electronic mail (mailto). The set of schemes is extensible, and new schemes are defined as new methods for accessing resources are introduced into the World Wide Web without any changes in the syntax of existing schemes.

Site

The site component specifies the Internet host containing the resource. It directly follows double slashes following the

scheme and its colon and ends at the next slash. It is composed of four parts, a user part followed by password, host, and port parts:

⟨user⟩:⟨password⟩@⟨host⟩:⟨port⟩

Access control on Internet hosts is often managed through user accounts, and some schemes use these accounts to gain access to a host. The user part of the site component names the account on the host through which the resource is accessed, and the password part specifies the password used to log in to the user account. They are separated by a colon.

Because the password to the user account is used without any form of encryption, its use poses a serious security risk. The use of the user and password parts is therefore strongly discouraged, with one exception. FTP servers often allow universal access via the account "anonymous", and it is common usage to give the electronic mail address of the user as the password. As a result, URLs using the FTP scheme employ "anonymous:email" as the user and password parts of the site component.

The host part identifies the Internet host containing the resource and is separated from the user and password parts by the commercial at sign "@". It is either the domain name of the host (9), such as "www.w3.org", or the Internet Protocol (10) address of the host, such as "18.23.0.23."

Each scheme accesses resources on a host with a different service that understands how to communicate by that scheme. These services are identified on a host by a port number, and these port numbers form the port part of a site component. The port follows the host part and is separated from it by a colon. For example, the URL "http://www.w3.org:80" indicates that the HTTP service should be accessed at port 80 on the host "www.w3.org."

The ports of services are often well-defined across hosts (11). As a result, port numbers are not typically specified because they can be guessed according to the scheme used to access a resource. For example, HTTP services are accessed by default at port 80 on a host, so that the previous URL specified as "http://www.w3.org" locates the same resource (assuming that the HTTP service on the host is communicating by port 80).

Not all parts of the site component are required when used. For example, the "mailto" scheme uses only the user and host parts, the "http" scheme uses the host and port parts, and the "ftp" and "telnet" schemes use all parts.

Path

The path component specifies the location of the resource on the host. It begins with the slash that ends the site component on the left and ends either at the end of the string or the optional query or fragment components on the right. For example, in the URL "http://www.w3.org/Addressing/schemes," the path component is "/Addressing/schemes."

The path component is composed of segments separated by a forward slash "/", giving the path component the look of a Unix-based filename. And, as with filenames, the path segments provide a hierarchy to the path component. However, it should be emphasized that the path component is not a filename. A scheme or site often chooses to interpret some or all of the path component as a filename, but doing so is an optimization and convenience rather than a requirement.

Query

The query component is a string of data with meaning only to the resource named by the preceding components of the URL. If specified, the query follows a question-mark "?" that ends the path component. For example, in the hypothetical URL "http://weather-service.org/forecast?today", the query component is "today" and has meaning only to the "forecast" resource.

When a resource is located by the components preceding the query component, the server gives it the query as input. Then the resource can return information based on the data in the query when it is accessed. In the previous example, the "forecast" resource could return the weather forecast specified by the query component, in this case, today's weather forecast.

Fragment

Technically, the fragment component is not part of a URL, although it is often transcribed and displayed as if it is. A fragment is an identifier that follows a URL and is separated from it by a crosshatched character "#." For example, the string "http://www.w3.org/Addressing/#background" has two elements, the URL "http://www.w3.org/Addressing/" and the fragment "background."

A fragment identifies a subcomponent of the object returned when the resource named by the URL is accessed. It is interpreted by the agent that accessed the URL once the access has completed. Although the format and interpretation of fragments depends on the object returned by a resource, they are commonly used to identify "anchors" placed inside of HTML documents.

An anchor names key components of an HTML document, such as section headers, tables, and images. URLs refer to such anchors by first specifying the location of the document and then using the anchor name as the fragment component. When processing these URLs, Web browsers first remove the anchor fragment to obtain the true URL of the document resource, use the true URL to retrieve the document, and then parse the document to find the anchor and position the viewing window at the anchor. In the previous example URL, a Web browser locates, downloads, and displays the document specified by the URL "http://www.w3.org/Addressing/", and then positions the document inside the browser window so that the section "background" is visible.

RELATIVE URLS

It is often convenient to locate resources relative to other resources. For example, hypertext documents located at one site typically refer to other documents at that site. Rather than having to specify the site in all of the references that link the documents together, it is more convenient for an author of a document to be able to locate other documents by using relative path components alone. Such relative naming is a very useful mechanism because it makes hypertext documents independent of path, site, and scheme.

A URL used to locate a resource relative to another is called a relative URL (12). A relative URL has one of three forms: "network", "site", and "path." A network-relative URL begins with two slash characters and is independent of the

scheme used to locate the URL. Site-relative URLs begin with one slash character and are independent of both the scheme and the site used to locate the URL. Path-relative URLs begin with a path segment and are independent of the scheme, site, and a prefix path. Relative URLs with path components also use the special path segments "." and ".." to refer to the current hierarchical level and next hierarchical level above the current level, respectively. These segments correspond to similar conventions commonly used in filenames.

For example, the following network-relative URL is independent of scheme:

```
//www.w3.org/Addressing/URL/Overview.html
```

If the site "www.w3.org" is running both the HTTP and FTP services and these services have matching path structures, then either service can be used to locate the resource. Because network-relative URLs are bound to a particular site and because it is uncommon for services to have matching path structures, these URLs are rarely used.

Removing the site component makes it a site-relative URL:

```
/Addressing/URL/Overview.html
```

Resources that use this site-relative URL can be moved from one site to another, along with this resource, without having to be changed. Site-relative URLs are often used when a group of resources on a site is shared by many other resources on that site and are more common than network-relative URLs.

Removing the path prefix "/Addressing/" now makes it a path-relative URL:

```
URL/Overview.html
```

Resources that use this URL, together with the resource named by the URL, can be moved anywhere in the path hierarchy on a site without having to be changed. They are the most common relative URLs because they are the most useful. For example, a collection of HTML documents and images on a particular subject is typically stored as a set of files in a directory subtree. The author of these documents and images links them with relative URLs that correspond to the relative directory structure storing the documents. By doing so, the author can later move the entire subtree from one directory in the file system to another, or from one machine to another, without having to change any of the URLs used to link the documents.

Determining Base URLs

A relative URL has meaning in only a particular context, and this context is determined by a "base URL." Then a relative URL and a base URL can be combined to form an absolute URL, completely naming the location of a particular resource.

The agent parsing a relative URL determines a base URL in one of four ways, described in order of precedence. First, in some types of documents the author embeds the base URL in the document content, such as HTML documents. Alternatively, message headers sent with documents specify base URLs, such as MIME message headers (13). Second, if no base URL is embedded in the document, then the document is examined to determine if it is encapsulated in a larger document as part of a composite media type (such as the "multipart/*" and "message/*" types defined by MIME (14)). If it is, then the base URL embedded in the encapsulating document is used. Third, if there is no encapsulating document or it does not specify a base URL, then the agent uses the absolute URL used to retrieve the document as the base URL. Lastly, if the agent cannot determine a base URL, then the base URL is considered the empty string, and the relative URL is interpreted as an absolute URL.

RELATED NAMING SCHEMES

This article has described how URLs are used to name the location of resources in the World Wide Web. However, they are not the only resource names used in the Web. Any name that identifies a resource on the Web is called a uniform resource identifier (URI) (15). URLs are a subset of all URIs, those URIs that identify resources by location. Another subset of URIs, called uniform resource names (URN) (16), are resource names without the location of the resource as part of the name.

A name that specifies location, such as a URL, has both advantages and drawbacks. Because the location of a resource is encoded in the name, knowing the name is enough to locate the resource. However, if the resource changes location (e.g., moves from one directory to another or from one server to another), then all uses of the URL for the previous location become invalid. The consequence of using URLs to refer to resources is that all references to the resource must be updated whenever the resource changes location. Performing these updates is a time-consuming, tedious, and error-prone process, leading to so-called "dangling URLs."

In contrast, URNs provide persistent, location-independent names for resources. Instead of naming the location of a resource, a URN names a unique, abstract identity corresponding to the resource. When a resource is accessed with a URN, the URN is first mapped to a URL naming the current location of the resource, and then this URL is used to find the resource. If the resource changes location, then subsequent uses of the URN map to a different URL that names the new location. URLs are currently the most commonly used names in the Web, but, as mechanisms for mapping URNs to URLs are developed and deployed, URNs will start to supplant the use of URLs.

BIBLIOGRAPHY

1. T. Berners–Lee et al., World-Wide Web: The information universe, in *Electron. Netw.: Res., Appl., Policy,* **2** (1): 52–58, 1992.
2. T. Berners–Lee, L. Masinter, and M. McCahill (eds.), *Uniform Resource Locators (URL),* RFC 1738, CERN, Xerox Corporation, Univ. Minnesota, 1994.
3. J. Gosling, B. Joy, and G. Steele, *The Java Language Specification,* Reading, MA: Addison-Wesley, 1996.
4. Netscape Communications Corporation, *Netscape Navigator,* [Online]. Available http://www.netscape.com.
5. Microsoft Corporation, *Internet Explorer,* [Online]. Available http://www.microsoft.com.
6. J. Postel and J. Reynolds, *File Transfer Protocol (FTP),* STD 9, RFC 959. Los Angeles, CA: USC/Information Sciences Institute, 1985.

7. R. Fielding et al., *Hypertext Transfer Protocol-HTTP/1.1.* RFC 2068, University of California, Irvine, Digital Equipment Corporation, MIT/LCS, 1997.
8. T. Berners–Lee and D. Connolly, *HyperText Markup Language Specification-2.0.* RFC 1866, MIT/LCS, November 1995.
9. P. Mockapetris, *Domain Names—Concepts and Facilities.* STD 13, RFC 1034, Los Angeles, CA: USC/Information Sciences Institute, November, 1987.
10. Information Sciences Institute, *Internet Protocol.* RFC 791, Los Angeles, CA: University of Southern California, 1981.
11. J. Reynolds and J. Postel, *Assigned Numbers.* RFC 1700, Los Angeles, CA: USC/Information Sciences Institute, 1994.
12. R. Fielding, *Relative Uniform Resource Locators,* RFC 1808. University of California, Irvine, June 1995.
13. N. Freed and N. Borenstein, *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies,* RFC 2045, Innosoft, First Virtual, 1996.
14. N. Freed and N. Borenstein, *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types,* RFC 2046, Innosoft, First Virtual, November 1996.
15. T. Berners-Lee, *Universal Resource Identifiers in WWW: A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as Used in the World-Wide Web,* RFC 1630, CERN, 1994.
16. R. Moats, *URN Syntax,* RFC 2141, AT&T, 1997.

GEOFFREY M. VOELKER
University of Washington

UNIX

UNIX is a general-purpose, interactive, time-sharing operating system originally invented in the 1970s at Bell Labs. There are many varieties of UNIX in current use, both free and commercial, but they all have well-integrated, standard, networking and graphics systems and provide a rich environment for programming, document preparation, and scientific computing. UNIX has been ported to more different types of hardware than any operating system in history (1). Because these systems have to function uniformly over networks populated by a bewildering variety of hardware types and different UNIX versions, the software is usually careful about providing hardware-independent binary data. As trends in computing hardware have changed from minicomputers, to vector supercomputers, to distributed networks of workstations and PCs, UNIX has evolved to meet the challenges.

IN THE BEGINNING

UNIX was originally invented in 1969 by Ken Thompson on a surplus DEC PDP-7 at AT&T's Bell Labs. It was modeled after the Multics operating system, which introduced many new concepts such as symmetric multiprocessing, a hierarchical file system, access control, and virtual memory. The name UNIX is a pun on Multics (Multiplexed Information and Computing Service), replacing "Multiplexed" by "Uniplexed," as it was originally just a simple single user system. Dennis Ritchie, who created the C programming language, joined Thompson and rewrote UNIX almost entirely in C during 1972 to 1974 (2–3). As far as we know, this made UNIX the first source-portable operating system in history. Undoubtedly the best single reference on the early days of UNIX is the famous July-August 1978 issue of the Bell System Technical Journal. This issue includes articles by Ritchie and Thompson (4) on the design and implementation of UNIX as well as a dozen others on the history of UNIX (5), the UNIX shell (6), text processing (7), language development tools, and more.

Computers in the 1970s were big, expensive systems with proprietary hardware and software. Vendors sought to lock customers into a particular family of machines. Command interfaces, system calls, etc. were complicated and uncorrelated from vendor to vendor. UNIX 7th edition (then called "Version 7") became available for DEC hardware (especially the popular 11/70 family of minicomputers) from AT&T in the late 1970s. The system came as a single tape, with all source, and for the cost of copying and issuing a license at about $300. The system came with no support, no user contacts, and no promises.

In the beginning UNIX was licensed almost for free to universities, including the full source code. This led to its immense popularity in academic environments. But with the release of Version 7, the licensing policy became more restrictive and the source code could no longer be used for teaching (1). One year later, in October 1980, BSD (Berkeley Software Distribution) UNIX 4.0 was released, again freely available to the public. It was based on the older free sources of UNIX and further developed at the University of California at Berkeley. It included many enhancements like paged virtual memory and TCP/IP networking. Several commercial versions were derived from this, including SunOS and ULTRIX (produced by Sun and DEC, respectively).

From the technical user's point of view, and apart from considerations of academic fashion, UNIX offered a number of very appealing features:

1. Every file is simply a stream of bytes. This sounds obvious in retrospect, but at the time, a typical operating system (OS) had all kinds of extra file attributes, such as record type and length. That complexity made it difficult for user programs to deal flexibly with files, particularly binary files of different types.
2. Devices are files. Each OS then had various utilities for doing input/output (IO) to devices. Merging devices into the file system had a number of beneficial results:
   - There were not as many system calls to remember, although the ioctl() system call expanded to take up some of the slack in that area.
   - Programs gained additional functionality (the ability to handle IO directly to/from devices) with little or no additional programming effort.
   - The number of specialized system utilities decreased accordingly.
3. It has a process model that is easy to use. The possibility to link the input and output streams of programs via pipes (buffers allowing asynchronous communication between processes under control of the kernel) had a great impact on how we write programs, even in technical work. It also led to quiet programs since unnecessary output could prevent a program from being used as a filter in a pipeline.

UNIX reduced the command interpreter, the shell, to just another user program, a change that much simpli-

fied the formal structure of the OS and led to the present proliferation of shells (overall a good thing). It also gave us convenient subprogram execution as a programming tool.

4. There was an inspired set of utilities. The core set was a then-unusual group of single-purpose programs such as sed, uniq, and tr. These made it possible to write the first spelling checker as just a pipeline of existing utilities, to serve as excellent examples to utility writers (which, in the end, much of UNIX programming is), and to give rise to the first entry of the trilogy of UNIX programming methods used to attack a problem—write a shell script, write a program, write a language.

   The distribution also included trof/eqn/tbl for typesetting (in fact, the famous 1978 issue of the Bell System Technical Journal was typeset using troff), as well as lex, a lexical analyzer, and yacc, a parser generator.

The possibilities contained in the UNIX utility set took a while to comprehend (because they were so different in structure from the large multifunction utilities with other operating systems), but once seen, they were inspiring and empowering.

The emergence of this powerful, and accessible bundle of capabilities showed a future rich with possibilities we had never seen before: an end to the mind-numbing proliferation of operating systems and their utilities and the emergence of a powerful, simple, user-oriented computing environment. All of this came about because the times were ready, and Bell Labs had one of those clusters of brilliant people that occur from time to time in every field. It was also because the international UNIX community had created a large and growing pool of freely-available software that will typically run with only minor changes on almost any UNIX system. This pool includes all of the components for software development (mostly from the efforts of the Free Software Foundation, http://www.fsf.org/, which is an independent group promoting the proliferation of free software with open source code) and some very powerful data management, manipulation, and display programs.

The current operating system is many times larger than Version 7 (mostly, for good reason), but its basic design and power are intact. Today, much of what has been described as a UNIX system has been fixed in the POSIX (Portable Operating System) standards, further improving portability. These IEEE standards define a UNIX-like interface (8), the shell and utilities, and real-time extensions (9).

In 1992, another major branch came into existence: Linux. It started as the hobby project of Finnish student, Linus Torvalds. Torvalds created a UNIX-like kernel for the Intel 80386 processor and released it to the public on the Internet. Others subsequently extended this kernel, drawing on the vast software resources from the UNIX world. By using the software which had been developed by the Free Software Foundation's GNU project and adding on the X-Windows interface provided by the XFree86 project, Linux was turned into a full featured major new UNIX system. The original UNIX tapped a rich vein of discontent and bottled-up creativity among technical users. Linux has done something similar in addition to capitalizing on the work of the Free Software Foundation. Today GNU/Linux has been ported to a wide variety of hardware and has turned out to be one of the most rapidly evolving operating systems ever.

For the price of a paper-back book, you can get the complete operating system, with all the bells and whistles (the kernel and all utilities, the X11 windowing environment, text processing software, compilers/debuggers, editors, file managers, and so on), along with all of the source code. This allows anyone to take cheap, ubiquitous hardware and build robust and reliable multiuser workstations that don't suffer the drawbacks of many commercial PC-based operating systems (frequent reboots, poor scheduling under load, weak security, and a single-user mentality). This democratization of desktop computing may ultimately be the most enduring legacy of the experiment begun at Bell Labs over 30 years ago.

CONCEPTS

Kernel

The kernel is the heart of an operating system. It provides the minimum functionality which defines it. Everything else is an add-on provided by external programs. One can, therefore, say that the kernel *is* the operating system.

The kernel provides the lowest abstraction layer of the underlying hardware. It is responsible for resource management and process scheduling. Traditionally, UNIX kernels included the device drivers for the hardware, all networking code, and all filesystem code. A change of drivers usually required recompiling the kernel and rebooting the system. Recent UNIX kernels are modular, so parts of them can be exchanged at runtime without rebooting.

Kernel Structure

One distinguishes between monolithic kernels and kernels with a message passing architecture. A monolithic kernel is compiled into one large binary file in which all its parts are accessible to all other parts, so that one kernel function can easily call any other one. While this provides minimum overhead, it can be unstructured and inflexible.

In a kernel with a message passing architecture, the kernel functions do not call each other directly, but send messages to each other, asking for certain operations to be performed. The same applies to user mode programs calling the kernel; the actual kernel call is performed by a library function which constructs the message required and passes it on to the kernel. This creates additional overhead and, therefore, is potentially slower than a monolithic kernel. Strictly speaking, a kernel with a message passing architecture could be monolithic as well, by being compiled into one binary, but still using messages to communicate between its parts. However, this makes relatively little sense, for it does not take full advantage of the message passing architecture.

The main advantage of a message passing architecture is that it can easily be split up and that operations can be delegated to external programs, device drivers, or other modules loaded at runtime. This makes it possible to build distributed systems that cooperate across multiple networked computers.

Microkernels

A common implementation of the message passing architecture is the microkernel. A microkernel provides only the mini-

mum functionality of task switching and memory management. Everything else, including device IO and filesystems, is handled by external processes, so-called servers, which run outside the kernel. This improves flexibility, as servers can be changed or restarted at any time. It also improves security because the servers do not necessarily run in kernel mode but can run as normal user mode processes with fewer privileges.

All communication with these servers is done through the kernel's message passing system which routes the messages to the appropriate server. Such a microkernel is extremely small and easy to port to different hardware architectures. A popular example of this is the MACH microkernel developed at Carnegie-Mellon University (MACH homepage: http://www.cs.cmu.edu/afs/cs.cmu.edu/project/mach/public/www/mach.html).

As most of the system's functionality which defines its API (Application Program Interface) is provided by processes running outside the microkernel, such a system can provide different operating system personalities. This is different from emulation, where one system gets simulated by another one. A microkernel can truly run multiple operating systems on the same hardware at the same time.

Hardware Abstraction

Normal programs run in user mode and have no direct access to the hardware. It is only through the kernel, which runs with special privileges, that they can access the hardware. To do so, they call the kernel to perform the required operation. The changes between user mode and kernel mode are called context switches and are generally quite expensive in terms of computation time. Every context switch involves saving all the processor registers, passing the parameters to the kernel, and calling the kernel function. To avoid unnecessary context switches, programs frequently use buffered IO as provided by user level libraries.

Processes

A process is an executing program, including its program counter, CPU registers, and variables. On a UNIX system, each process has its own virtual CPU and address space. The kernel is responsible for switching the existing CPU(s) between the waiting processes. As UNIX provides preemptive multitasking (as opposed to cooperative multitasking), the processes do not need to explicitly release the CPU but get scheduled by the kernel. Processes carry different priorities which allow the scheduler to assign them more or less CPU time.

All processes are related by a parent-child relationship. The only way to create a new process is by using the fork() system call which makes an identical copy of the current process. To start a second program, a process needs to call fork(), and then, one of the two copies needs to replace itself with the new program by means of the exec() call. As a consequence of this, each process has a parent process and the relationship between processes is a tree-like structure with the **init** process at its root. The init process gets started by the kernel at boot time and is responsible for system initialization and boot-up.

The kernel maintains a process table with information on all processes, including the list of open files, the state (running, waiting, stopped, sleeping, etc.), the program counter, the stack pointer, the memory allocation map, working directory, user ID, group ID, priority, and parent process ID.

While processes are generally independent of each other, there are mechanisms for them to communicate with each other. The simplest of these are to send each other signals or to send data through pipes. Signals interrupt the normal flow of a process and force it into a special signal-handling routing to react to it, before continuing normal operation. Signals can be sent between any two processes belonging to the same user. Pipes can only exist between processes sharing a common parent or having a parent/child relationship. While signals just allow to tell the other process that the signal has been sent (i.e., no additional information can be transmitted), pipes allow full communication between the two processes, sending any kind of data through them. Another important one is the possibility of sharing memory between different processes. For this, they register a common area of physical memory to be shared. To control access to shared resources, UNIX supports a control mechanism invented by Dijkstra known as semaphores.

A concept similar to a process is a thread. Every process consists of at least one thread which can be thought of as the currently executed code of the process together with its stack. A process can create additional threads which can execute in parallel, each of them having their own stack but sharing the same address space and resources. Programs using multiple threads provide concurrent execution of functions, without the large overhead of creating multiple processes, but have to be written carefully to avoid problems caused by multiple threads accessing the same memory.

INFLUENCE ON OTHER OPERATING SYSTEMS

UNIX has pioneered many concepts now commonly found in other operating systems. Its history is closely linked to the C programming language, and it has been the predominant operating system that introduced networking and on which the Internet has been built. Many of these concepts have made their way into other systems. Today, the TCP/IP networking protocol has become the *de facto* standard across platforms. The BSD UNIX socket interface to network programming has been adopted by other systems, with the Windows Winsock interface being a prominent example. Other operting systems like Windows NT are becoming more and more UNIX-like and widely conforming to the POSIX standards.

BIBLIOGRAPHY

1. A. S. Tanenbaum, *Operating Systems Design and Implementation,* Englewood Cliffs, NJ: Prentice-Hall, 1987.
2. S. C. Johnson and D. M. Ritchie, Portability of C programs and the UNIX system, *Bell Syst. Tech. J.,* **57** (6): 2021–2048, 1978.
3. D. M. Ritchie et al., The C programming language, *Bell Syst. Tech. J.,* **57** (6): 1991–2020, 1978.
4. D. M. Ritchie and K. Thompson, The UNIX time-sharing system, *Bell Syst. Tech. J.,* **57** (6): 1905–1930, 1978.
5. M. D. McIlroy, E. N. Pinson, and B. A. Tague, Forward, *Bell Syst. Tech. J.,* **57** (6): 1899–1904, 1978.
6. S. R. Bourne, The UNIX shell, *Bell Syst. Tech. J.,* **57** (6): 1971–1990, 1978.

7. B. W. Kernighan, M. E. Lesk, and J. F. Ossana, Jr., Document preparation, *Bell Syst. Tech. J.,* **57** (6): 2115–2136, 1978.
8. D. Lewine, *POSIX Programmer's Guide,* O'Reilly & Associates, 1991.
9. B. Gallmeister, *POSIX.4, Programming for the Real World,* O'Reilly & Associates, 1995.

KARSTEN BALLÜDER
Heriot-Watt University

JOHN A. SCALES
Colorado School of Mines

MARTIN L. SMITH
New England Research

UNSTABLE OPERATING POINT. See CIRCUIT STABILITY OF DC OPERATING POINTS.

URL. See UNIVERSAL RESOURCE LOCATOR.