**Message Queues**

Message queues allow one or more processes to write messages that will be read by one or more reading processes. Linux maintains a list of message queues, the msgque vector: each element of which points to a msqid\_ds data structure that fully describes the message queue. When message queues are created, a new msqid\_ds data structure is allocated from system memory and inserted into the vector.

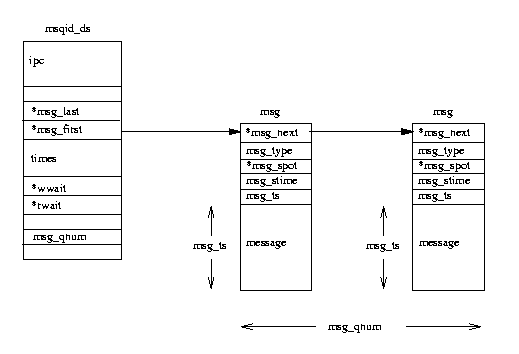


Figure: System V IPC Message Queues

Each msqid\_ds data structure contains an ipc\_perm data structure and pointers to the messages entered onto this queue. In addition, Linux keeps queue modification times such as the last time that this queue was written to and so on. Themsqid\_ds also contains two wait queues: one for the writers to the queue and one for the readers of the queue.

Each time a process attempts to write a message to the write queue, its effective user and group identifiers are compared with the mode in this queue's ipc\_perm data structure. If the process can write to the queue then the message may be copied from the process' address space into a msg data structure and put at the end of this message queue. Each message is tagged with an application specific type, agreed between the cooperating processes. However, there may be no room for the message as Linux restricts the number and length of messages that can be written. In this case the process will be added to this message queue's write wait queue and the scheduler will be called to select a new process to run. It will be awakened when one or more messages have been read from this message queue.

Reading from the queue is similar. Again, the process' access rights to the write queue are checked. A reading process may choose to either get the first message in the queue regardless of its type or select messages with particular types. If no messages match this criteria the reading process will be added to the message queue's read wait queue and the scheduler run. When a new message is written to the queue this process will be awakened and run again.

**SEMAPHORE**

In its simplest form a semaphore is a location in memory whose value can be tested and set by more than one process. The test and set operation is, so far as each process is concerned, uninterruptible or atomic; once started nothing can stop it. The result of the test and set operation is the addition of the current value of the semaphore and the set value, which can be positive or negative. Depending on the result of the test and set operation one process may have to sleep until the semphore's value is changed by another process. Semaphores can be used to implement critical regions, areas of critical code that only one process at a time should be executing.

Although a program variable could be considered "a location in memory whose value can be tested and set", the key different is that with a semaphore is accessible to other processes, whereas a variable is only accessible to the one process that created it. The fact that it is accessible from multiple processes is the key feature of a semaphore.

Say you had many cooperating processes reading records from and writing records to a single data file. You would want that file access to be strictly coordinated. You could use a semaphore with an initial value of 1 and, around the file operating code, put two semaphore operations, the first to test and decrement the semaphore's value and the second to test and increment it. The first process to access the file would try to decrement the semaphore's value and it would succeed, the semaphore's value now being 0. This process can now go ahead and use the data file but if another process wishing to use it now tries to decrement the semaphore's value it would fail as the result would be -1. That process will be suspended until the first process has finished with the data file. When the first process has finished with the data file it will increment the semaphore's value, making it 1 again. Now the waiting process can be awakened and this time its attempt to decrement the semaphore will succeed.

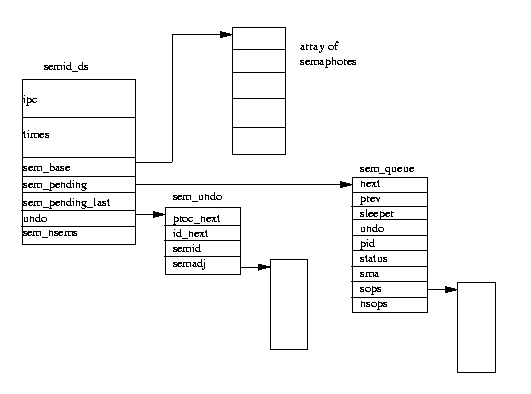


Figure: System V IPC Semaphores

System V IPC semaphore objects each describe a semaphore array and Linux uses the semid\_ds data structure to represent this. All of the semid\_ds data structures in the system are pointed at by the semary, a vector of pointers. There are sem\_nsems in each semaphore array, each one described by a sem data structure pointed at by sem\_base. All of the processes that are allowed to manipulate the semaphore array of a System V IPC semaphore object may make system calls that perform operations on them. The system call can specify many operations and each operation is described by three inputs: the semaphore index, the operation value and a set of flags. The semaphore index is an index into the semaphore array and the operation value is a numerical value that will be added to the current value of the semaphore. First Linux tests whether or not all of the operations would succeed. An operation will succeed if the operation value added to the semaphore's current value would be greater than zero or if both the operation value and the semaphore's current value are zero. If any of the semaphore operations would fail Linux may suspend the process but only if the operation flags have not requested that the system call is non-blocking. If the process is to be suspended then Linux must save the state of the semaphore operations to be performed and put the current process onto a wait queue. It does this by building asem\_queue data structure on the stack and filling it out. The new sem\_queue data structure is put at the end of this semaphore object's wait queue (using the sem\_pending and sem\_pending\_last pointers). The current process is put on the wait queue in the sem\_queue data structure (sleeper) and the scheduler called to choose another process to run.

If all of the semaphore operations would have succeeded and the current process does not need to be suspended, Linux goes ahead and applies the operations to the appropriate members of the semaphore array. Now Linux must check that any waiting, suspended, processes may now apply their semaphore operations. It looks at each member of the operations pending queue (sem\_pending) in turn, testing to see if the semphore operations will succeed this time. If they will then it removes the sem\_queue data structure from the operations pending list and applies the semaphore operations to the semaphore array. It wakes up the sleeping process making it available to be restarted the next time the scheduler runs. Linux keeps looking through the pending list from the start until there is a pass where no semaphore operations can be applied and so no more processes can be awakened.

There is a problem with semaphores: deadlocks. These occur when one process has altered the semaphore's value as it enters a critical region but then fails to leave the critical region because it crashed or was killed. Linux protects against this by maintaining lists of adjustments to the semaphore arrays. The idea is that when these adjustments are applied, the semaphores will be put back to the state that they were in before the a process' set of semaphore operations were applied. These adjustments are kept in sem\_undo data structures queued both on the semid\_ds data structure and on thetask\_struct data structure for the processes using these semaphore arrays.

Each individual semaphore operation may request that an adjustment be maintained. Linux will maintain at most onesem\_undo data structure per process for each semaphore array. If the requesting process does not have one, then one is created when it is needed. The new sem\_undo data structure is queued both onto this process' task\_struct data structure and onto the semaphore array's semid\_ds data structure. As operations are applied to the semphores in the semaphore array the negation of the operation value is added to this semphore's entry in the adjustment array of this process'sem\_undo data structure. So, if the operation value is 2, then -2 is added to the adjustment entry for this semaphore.

When processes are deleted, as they exit Linux works through their set of sem\_undo data structures applying the adjustments to the semaphore arrays. If a semaphore set is deleted, the sem\_undo data structures are left queued on the process' task\_struct but the semaphore array identifier is made invalid. In this case the semaphore clean up code simply discards the sem\_undo data structure.

**Named semaphores**

A named semaphore is identified by a name of the form */somename*. Two processes can operate on the same named semaphore by passing the same name to[***sem\_open***](http://linux.die.net/man/3/sem_open)*(3)*.

The ***[sem\_open](http://linux.die.net/man/3/sem_open)****(3)* function creates a new named semaphore or opens an existing named semaphore. After the semaphore has been opened, it can be operated on using ***[sem\_post](http://linux.die.net/man/3/sem_post)****(3)* and ***[sem\_wait](http://linux.die.net/man/3/sem_wait)****(3)*. When a process has finished using the semaphore, it can use ***[sem\_close](http://linux.die.net/man/3/sem_close)****(3)* to close the semaphore. When all processes have finished using the semaphore, it can be removed from the system using ***[sem\_unlink](http://linux.die.net/man/3/sem_unlink)****(3)*.

**Unnamed semaphores (memory-based semaphores)**

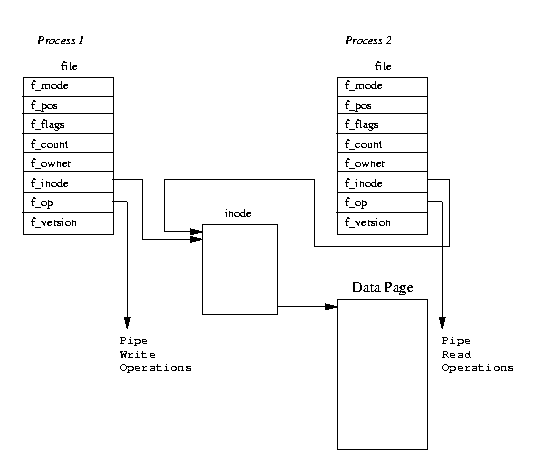
An unnamed semaphore does not have a name. Instead the semaphore is placed in a region of memory that is shared between multiple threads (a *thread-shared semaphore*) or processes (a *process-shared semaphore*). A thread-shared semaphore is placed in an area of memory shared between by the threads of a process, for example, a global variable

**Pipes**

The common Linux shells all allow redirection. For example

$ ls | pr | lpr

pipes the output from the ls command listing the directory's files into the standard input of the pr command which paginates them. Finally the standard output from the pr command is piped into the standard input of the lpr command which prints the results on the default printer. Pipes then are unidirectional byte streams which connect the standard output from one process into the standard input of another process. Neither process is aware of this redirection and behaves just as it would normally. It is the shell that sets up these temporary pipes between the processes.

  
Figure: Pipes

In Linux, a pipe is implemented using two file data structures which both point at the same temporary VFS inode which, in turn, points at a physical page within memory. Figure [5.1](http://www.linux-tutorial.info/modules.php?name=MContent&pageid=290#pipes-figure) shows that each file data structure contains pointers to different file operation routine vectors: one for writing to the pipe, the other for reading from the pipe.

This hides the underlying differences from the generic system calls which read and write to ordinary files. As the writing process writes to the pipe, bytes are copied into the shared data page and when the reading process reads from the pipe, bytes are copied from the shared data page. Linux must synchronize access to the pipe. It must make sure that the reader and the writer of the pipe are in step and to do this it uses locks, wait queues and signals.

When the writer wants to write to the pipe it uses the standard write library functions. These all pass file descriptors that are indices into the process' set of file data structures, each one representing an open file or, as in this case, an open pipe. The Linux system call uses the write routine pointed at by the file data structure describing this pipe. That write routine uses information held in the VFS inode representing the pipe to manage the write request.

If there is enough room to write all of the bytes into the pipe and, so long as the pipe is not locked by its reader, Linux locks it for the writer and copies the bytes to be written from the process' address space into the shared data page. If the pipe is locked by the reader or if there is not enough room for the data then the current process is made to sleep on the pipe inode's wait queue and the scheduler is called so that another process can run. It is interruptible, so it can receive signals and it will be awakened by the reader when there is enough room for the write data or when the pipe is unlocked. When the data has been written, the pipe's VFS inode is unlocked and any waiting readers sleeping on the inode's wait queue will themselves be awakened.

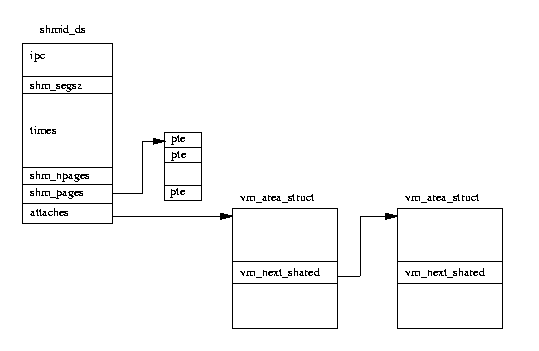
Reading data from the pipe is a very similar process to writing to it.

Processes are allowed to do non-blocking reads (depending on the mode in which they opened the file or pipe and if there is no data to be read or if the pipe is locked, an error will be returned,as in this case). This means that the process *can*continue to run. The alternative is to wait on the pipe inode's wait queue until the write process has finished. When both processes have finished with the pipe, the pipe inode is discarded along with the shared data page.

Linux also supports *named* pipes, also known as FIFOs because pipes operate on a First In, First Out principle. The first data written into the pipe is the first data read from the pipe. Unlike pipes, FIFOs are not temporary objects, they are entities in the file system and can be created using the mkfifo command. Processes are free to use a FIFO so long as they have appropriate access rights to it. The way that FIFOs are opened is a little different from pipes. A pipe (its two filedata structures, its VFS inode and the shared data page) is created in one go whereas a FIFO already exists and is opened and closed by its users. Linux must handle readers opening the FIFO before writers open it as well as readers reading before any writers have written to it. That aside, FIFOs are handled almost exactly the same way as pipes and they use the same data structures and operations.

**Shared Memory**

Shared memory allows one or more processes to communicate via memory that appears in all of their virtual address spaces. The pages of the virtual memory is referenced by page table entries in each of the sharing processes' page tables. It does not have to be at the same address in all of the processes' virtual memory. As with all System V IPC objects, access to shared memory areas is controlled via keys and access rights checking. Once the memory is being shared, there are no checks on how the processes use it. They must rely on other mechanisms, for example System V semaphores, to synchronize access to the memory.



System V IPC Shared Memory

Each newly created shared memory area is represented by a shmid\_ds data structure. These are kept in the shm\_segsvector.

The shmid\_ds data structure decribes how big the area of shared memory is, how many processes are using it and information about how that shared memory is mapped into their address spaces. It is the creator of the shared memory that controls the access permissions to that memory and whether its key is public or private. If it has enough access rights it may also lock the shared memory into physical memory.

Each process that wishes to share the memory must attach to that virtual memory via a system call. This creates a newvm\_area\_struct data structure describing the shared memory for this process. The process can choose where in its virtual address space the shared memory goes or it can let Linux choose a free area large enough. The new vm\_area\_structstructure is put into the list of vm\_area\_struct pointed at by the shmid\_ds. The vm\_next\_shared and vm\_prev\_sharedpointers are used to link them together. The virtual memory is not actually created during the attachment; it happens when the first process attempts to access it.

The first time that a process accesses one of the pages of the shared virtual memory, a page fault will occur. When Linux fixes up that page fault it finds the vm\_area\_struct data structure describing it. This contains pointers to handler routines for this type of shared virtual memory. The shared memory page fault handling code looks in the list of page table entries for this shmid\_ds to see if one exists for this page of the shared virtual memory. If it does not exist, it will allocate a physical page and create a page table entry for it.

This entry is saved in the current process' page tables and the shmid\_ds.. Consequently, when the next process that attempts to access this memory gets a page fault, the shared memory fault handling code will use this newly created physical page for that process too. So, the first process that accesses a page of the shared memory causes it to be created and thereafter access by the other processes cause that page to be added into their virtual address spaces.

When processes no longer wish to share the virtual memory, they detach from it. So long as other processes are still using the memory the detach only affects the current process. Its vm\_area\_struct is removed from the shmid\_ds data structure and deallocated. The current process's page tables are updated to invalidate the area of virtual memory that it once shared. When the last process sharing the memory detaches from it, the pages of the shared memory current in physical memory are freed, as is the shmid\_ds data structure for this shared memory.

Further complications arise when shared virtual memory is not locked into physical memory. In this case the pages of the shared memory may be swapped out to the system's swap disk during periods of high memory usage.

## What is Perl?

Perl is a general-purpose programming language originally developed for text manipulation and now used for a wide range of tasks including system administration, web development, network programming, GUI development, and more.

The language is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal). Its major features are that it's easy to use, supports both procedural and object-oriented (OO) programming, has powerful built-in support for text processing, and has one of the world's most impressive collections of third-party modules.

Running Perl programs

To run a Perl program from the Unix command line:

1. perl progname.pl

Alternatively, put this as the first line of your script:

1. #!/usr/bin/env perl

... and run the script as /path/to/script.pl. Of course, it'll need to be executable first, so chmod 755 script.pl (under Unix).

## Basic syntax overview

A Perl script or program consists of one or more statements. These statements are simply written in the script in a straightforward fashion. There is no need to have a main() function or anything of that kind.

Perl statements end in a semi-colon:

1. [**print**](http://perldoc.perl.org/functions/print.html) "Hello, world";

Comments start with a hash symbol and run to the end of the line

1. # This is a comment

Whitespace is irrelevant:

1. [**print**](http://perldoc.perl.org/functions/print.html)
2. "Hello, world"
3. ;

... except inside quoted strings:

1. # this would print with a linebreak in the middle
2. [**print**](http://perldoc.perl.org/functions/print.html) "Hello
3. world";

Double quotes or single quotes may be used around literal strings:

1. [**print**](http://perldoc.perl.org/functions/print.html) "Hello, world";
2. [**print**](http://perldoc.perl.org/functions/print.html) 'Hello, world';

However, only double quotes "interpolate" variables and special characters such as newlines (\n ):

1. [**print**](http://perldoc.perl.org/functions/print.html) "Hello, $name\n"; # works fine
2. [**print**](http://perldoc.perl.org/functions/print.html) 'Hello, $name\n'; # prints $name\n literally

Numbers don't need quotes around them:

1. [**print**](http://perldoc.perl.org/functions/print.html) 42;

You can use parentheses for functions' arguments or omit them according to your personal taste. They are only required occasionally to clarify issues of precedence.

1. [**print**](http://perldoc.perl.org/functions/print.html)("Hello, world\n");
2. [**print**](http://perldoc.perl.org/functions/print.html) "Hello, world\n";

More detailed information about Perl syntax can be found in [perlsyn](http://perldoc.perl.org/perlsyn.html).

## Perl variable types

Perl has three main variable types: scalars, arrays, and hashes.

* **Scalars**

A scalar represents a single value:

* 1. [**my**](http://perldoc.perl.org/functions/my.html) $animal = "camel";
  2. [**my**](http://perldoc.perl.org/functions/my.html) $answer = 42;

Scalar values can be strings, integers or floating point numbers, and Perl will automatically convert between them as required. There is no need to pre-declare your variable types, but you have to declare them using the[**my**](http://perldoc.perl.org/functions/my.html) keyword the first time you use them. (This is one of the requirements of [**use**](http://perldoc.perl.org/functions/use.html) strict; .)

Scalar values can be used in various ways:

* 1. [**print**](http://perldoc.perl.org/functions/print.html) $animal;
  2. [**print**](http://perldoc.perl.org/functions/print.html) "The animal is $animal\n";
  3. [**print**](http://perldoc.perl.org/functions/print.html) "The square of $answer is ", $answer \* $answer, "\n";

There are a number of "magic" scalars with names that look like punctuation or line noise. These special variables are used for all kinds of purposes, and are documented in [perlvar](http://perldoc.perl.org/perlvar.html). The only one you need to know about for now is $\_ which is the "default variable". It's used as the default argument to a number of functions in Perl, and it's set implicitly by certain looping constructs.

* 1. [**print**](http://perldoc.perl.org/functions/print.html); # prints contents of $\_ by default
* **Arrays**

An array represents a list of values:

* 1. [**my**](http://perldoc.perl.org/functions/my.html) @animals = ("camel", "llama", "owl");
  2. [**my**](http://perldoc.perl.org/functions/my.html) @numbers = (23, 42, 69);
  3. [**my**](http://perldoc.perl.org/functions/my.html) @mixed = ("camel", 42, 1.23);

Arrays are zero-indexed. Here's how you get at elements in an array:

* 1. [**print**](http://perldoc.perl.org/functions/print.html) $animals[0]; # prints "camel"
  2. [**print**](http://perldoc.perl.org/functions/print.html) $animals[1]; # prints "llama"

The special variable $#array tells you the index of the last element of an array:

* 1. [**print**](http://perldoc.perl.org/functions/print.html) $mixed[$#mixed]; # last element, prints 1.23

You might be tempted to use $#array + 1 to tell you how many items there are in an array. Don't bother. As it happens, using @array where Perl expects to find a scalar value ("in scalar context") will give you the number of elements in the array:

* 1. if (@animals < 5) { ... }

The elements we're getting from the array start with a $ because we're getting just a single value out of the array; you ask for a scalar, you get a scalar.

To get multiple values from an array:

* 1. @animals[0,1]; # gives ("camel", "llama");
  2. @animals[0..2]; # gives ("camel", "llama", "owl");
  3. @animals[1..$#animals]; # gives all except the first element

This is called an "array slice".

You can do various useful things to lists:

* 1. [**my**](http://perldoc.perl.org/functions/my.html) @sorted = [**sort**](http://perldoc.perl.org/functions/sort.html) @animals;
  2. [**my**](http://perldoc.perl.org/functions/my.html) @backwards = [**reverse**](http://perldoc.perl.org/functions/reverse.html) @numbers;

There are a couple of special arrays too, such as @ARGV (the command line arguments to your script) and @\_(the arguments passed to a subroutine). These are documented in [perlvar](http://perldoc.perl.org/perlvar.html).

* **Hashes**

A hash represents a set of key/value pairs:

* 1. [**my**](http://perldoc.perl.org/functions/my.html) %fruit\_color = ("apple", "red", "banana", "yellow");

You can use whitespace and the => operator to lay them out more nicely:

* 1. [**my**](http://perldoc.perl.org/functions/my.html) %fruit\_color = (
  2. apple => "red",
  3. banana => "yellow",
  4. );

To get at hash elements:

* 1. $fruit\_color{"apple"}; # gives "red"

You can get at lists of keys and values with [**keys()**](http://perldoc.perl.org/functions/keys.html) and [**values()**](http://perldoc.perl.org/functions/values.html).

* 1. [**my**](http://perldoc.perl.org/functions/my.html) @fruits = [**keys**](http://perldoc.perl.org/functions/keys.html) %fruit\_colors;
  2. [**my**](http://perldoc.perl.org/functions/my.html) @colors = [**values**](http://perldoc.perl.org/functions/values.html) %fruit\_colors;

Hashes have no particular internal order, though you can sort the keys and loop through them.

Just like special scalars and arrays, there are also special hashes. The most well known of these is %ENVwhich contains environment variables. Read all about it (and other special variables) in [perlvar](http://perldoc.perl.org/perlvar.html).

Scalars, arrays and hashes are documented more fully in [perldata](http://perldoc.perl.org/perldata.html).

More complex data types can be constructed using references, which allow you to build lists and hashes within lists and hashes.

A reference is a scalar value and can refer to any other Perl data type. So by storing a reference as the value of an array or hash element, you can easily create lists and hashes within lists and hashes. The following example shows a 2 level hash of hash structure using anonymous hash references.

1. [**my**](http://perldoc.perl.org/functions/my.html) $variables = {
2. scalar => {
3. description => "single item",
4. sigil => '$',
5. },
6. array => {
7. description => "ordered list of items",
8. sigil => '@',
9. },
10. hash => {
11. description => "key/value pairs",
12. sigil => '%',
13. },
14. };
15. [**print**](http://perldoc.perl.org/functions/print.html) "Scalars begin with a $variables->{'scalar'}->{'sigil'}\n";

**CGI**

The **Common Gateway Interface** (**CGI**) is a standard (see [RFC3875: CGI Version 1.1](http://www.ietf.org/rfc/rfc3875.txt)) that defines how [webserver](http://en.wikipedia.org/wiki/Webserver" \o "Webserver) software can delegate the generation of [webpages](http://en.wikipedia.org/wiki/Webpage" \o "Webpage) to a [console application](http://en.wikipedia.org/wiki/Console_application). Such applications are known as *CGI scripts*; they can be written in any programming language, although [scripting languages](http://en.wikipedia.org/wiki/Scripting_language) are often used. In simple words the CGI provides an interface between the webservers and the clients. They will identify the request from client and will invoke appropriate function to return the result to the requested clients

## purpose

The task of a webserver is to respond to requests for webpages issued by [clients](http://en.wikipedia.org/wiki/Client_(computing)) (usually [web browsers](http://en.wikipedia.org/wiki/Web_browser)) by analyzing the content of the request (which is mostly in its [URL](http://en.wikipedia.org/wiki/URL)), determining an appropriate document to send in response, and returning it to the client.

If the request identifies a file on disk, the server can just return the file's contents. Alternatively, the document's content can be composed on the fly. One way of doing this is to let a console application compute the document's contents, and tell the web server to use that console application. CGI specifies which information is communicated between the webserver and such a console application, and how.

The webserver software will invoke the console application as a command. CGI defines how information about the request (such as the [URL](http://en.wikipedia.org/wiki/URL)) is passed to the command in the form of arguments and [environment variables](http://en.wikipedia.org/wiki/Environment_variable). The application is supposed to write the output document to [standard output](http://en.wikipedia.org/wiki/Standard_output); CGI defines how it can pass back extra information about the output (such as the [MIME type](http://en.wikipedia.org/wiki/MIME_type), which defines the type of document being returned) by prepending it with [headers](http://en.wikipedia.org/wiki/List_of_HTTP_headers).

## Drawbacks

Calling a command generally means the invocation of a newly created [process](http://en.wikipedia.org/wiki/Computer_process). Starting up the process can take up much more time and memory than the actual work of generating the output, especially when the program still needs to be [interpreted](http://en.wikipedia.org/wiki/Interpret) or [compiled](http://en.wikipedia.org/wiki/Compiler). If the command is called often, the resulting workload can quickly overwhelm the web server.

The overhead involved in interpretation may be reduced by using compiled CGI programs, such as those in [C](http://en.wikipedia.org/wiki/C_(programming_language))/[C++](http://en.wikipedia.org/wiki/C%2B%2B), rather than using [Perl](http://en.wikipedia.org/wiki/Perl) or other scripting languages. The overhead involved in process creation can be reduced by solutions such as [FastCGI](http://en.wikipedia.org/wiki/FastCGI" \o "FastCGI), or by running the application code entirely within the webserver using special extension modules.

**Tcl**

**Tcl** (originally from "Tool Command Language", but conventionally rendered as "Tcl" rather than "TCL"; pronounced as "[tickle](http://en.wiktionary.org/wiki/tickle)" or "tee-see-ell"[[2]](http://en.wikipedia.org/wiki/Tcl#cite_note-1)) is a [scripting language](http://en.wikipedia.org/wiki/Scripting_language) created by [John Ousterhout](http://en.wikipedia.org/wiki/John_Ousterhout).[[3]](http://en.wikipedia.org/wiki/Tcl#cite_note-2) Originally "born out of frustration,"[[4]](http://en.wikipedia.org/wiki/Tcl#cite_note-3) according to the author, with programmers devising their own (poor quality) languages intended to be embedded into applications, Tcl gained acceptance on its own. It is commonly used for [rapid prototyping](http://en.wikipedia.org/wiki/Rapid_prototyping), scripted applications, GUIs and testing. Tcl is used on embedded systems platforms, both in its full form and in several other small-footprinted versions. Tcl is also used for [CGI](http://en.wikipedia.org/wiki/Common_Gateway_Interface) scripting and as the scripting language for the [Eggdrop](http://en.wikipedia.org/wiki/Eggdrop" \o "Eggdrop) bot. Tcl is popularly used today in many automated test harnesses, both for software and hardware, and has a loyal following in the Network Testing and [SQA](http://en.wikipedia.org/wiki/Software_Quality_Assurance" \o "Software Quality Assurance)communities.

The combination of Tcl and the [Tk](http://en.wikipedia.org/wiki/Tk_(framework)" \o "Tk (framework)) [GUI toolkit](http://en.wikipedia.org/wiki/Widget_toolkit) is referred to as **Tcl/Tk** which is often pronounced "tickle tock."

## Features

Tcl's features include

* All operations are [commands](http://en.wikipedia.org/wiki/Command_(computing)), including language structures. They are written in [prefix notation](http://en.wikipedia.org/wiki/Prefix_notation).
* Commands are commonly [variadic](http://en.wikipedia.org/wiki/Variadic" \o "Variadic).
* Everything can be dynamically redefined and overridden.
* All [data types](http://en.wikipedia.org/wiki/Data_type) can be manipulated as [strings](http://en.wikipedia.org/wiki/String_(computer_science)), including [code](http://en.wikipedia.org/wiki/Code).
* [Event-driven interface](http://en.wikipedia.org/wiki/Event-driven_programming) to [sockets](http://en.wikipedia.org/wiki/Socket) and [files](http://en.wikipedia.org/wiki/Computer_file). Time-based and user-defined events are also possible.
* Variable visibility restricted to lexical (static) scope by default, but [uplevel](http://en.wikipedia.org/wiki/Uplevel" \o "Uplevel) and upvar allowing procs to interact with the enclosing functions' scopes.
* All commands defined by Tcl itself generate error messages on incorrect usage.
* [Extensible](http://en.wikipedia.org/wiki/Extensibility), via [C](http://en.wikipedia.org/wiki/C_(programming_language)), [C++](http://en.wikipedia.org/wiki/C%2B%2B), [Java](http://en.wikipedia.org/wiki/Java_(programming_language)), and Tcl.
* [Interpreted language](http://en.wikipedia.org/wiki/Interpreted_language) using [bytecode](http://en.wikipedia.org/wiki/Bytecode" \o "Bytecode)
* Full [Unicode](http://en.wikipedia.org/wiki/Unicode) (3.1) support, first released 1999.
* Platform independent: [Win32](http://en.wikipedia.org/wiki/Win32), [UNIX](http://en.wikipedia.org/wiki/Unix), [Linux](http://en.wikipedia.org/wiki/Linux), [Mac](http://en.wikipedia.org/wiki/Apple_Macintosh), etc.
* Close integration with windowing ([GUI](http://en.wikipedia.org/wiki/GUI)) interface [Tk](http://en.wikipedia.org/wiki/Tk_(framework)).
* Multiple distribution mechanisms exist:
  + Full development version (e. g. ActiveState Tcl)
  + tclkit (kind of single-file runtime, only about 1 megabyte in size)
  + starpack (single-file executable of a script/program, derived from the tclkit technology)
  + [BSD-licensed](http://en.wikipedia.org/wiki/BSD_licenses) freely distributable source.

## Syntax and Fundamental Semantics

A Tcl script consists of several command invocations. A command invocation is a list of words separated by whitespace and terminated by a newline or semicolon.

word0 word1 word2 ... wordN

The first word is the name of a command, which is not built into the language, but which is in the library. The following words are arguments. So we have:

commandName argument1 argument2 ... argumentN

Practical example, using the *puts* command which outputs a string, adding a trailing newline, by default to the *stdout* channel:

puts "Hello, world!"

Variables and the results of other commands can be substituted inside strings too, such as in this example where we use *set* and *expr* to store a calculation result in a variable, and *puts*to print the result together with some explanatory text:

*# Good style would put the expression (1+2+3+4+5, in this case) inside {curly braces}*

**set** sum [**expr** 1+2+3+4+5]

puts "The sum of the numbers 1..5 is $sum."

Formally, words are either written as-is, with double-quotes around them (allowing whitespace characters to be embedded), or with curly-brace characters around them, which suppresses all substitutions inside (except for backslash-newline elimination). In bare and double-quoted words, three types of substitution occur (once, in a single left-to-right scan through the word):

* **Command substitution** replaces the contents of balanced square brackets with the result of evaluating the script contained inside. For example, “**[expr 1+2+3]**” is replaced with the result of evaluating the contained expression (i.e. 6) since that's what the expr command does.
* **Variable substitution** replaces a dollar-sign followed by the name of a variable with the contents of the variable. For example, “**$foo**” is replaced with the contents of the variable called “foo”. The variable name may be surrounded in curly braces so as to delimit what is and isn't the variable name in otherwise ambiguous cases.
* **Backslash substitution** replaces a backslash followed by a letter with another character. For example, “**\n**” is replaced with a newline.

From Tcl 8.5 onwards, any word may be prefixed by “**{\*}**” to cause that word to be split apart into its constituent sub-words for the purposes of building the command invocation (similar to the “**,@**” sequence of [Lisp](http://en.wikipedia.org/wiki/Lisp_(programming_language))'s quasiquote feature).

As a consequence of these rules, the result of any command may be used as an argument to any other command. Also, there is no operator or command for string concatenation, as the language concatenates directly. Note that, unlike in [Unix command shells](http://en.wikipedia.org/wiki/Bourne_shell), Tcl does not reparse any string unless explicitly directed to do so, which makes interactive use more cumbersome but scripted use more predictable (e.g. the presence of spaces in filenames does not cause difficulties).

To summarize: there is one basic construct (the command) and a set of simple substitution rules. The single equality sign (=) for example is not used at all, and the double equality sign (==) is the test for equality, and even then only in expression contexts such as the expr command or the first argument to if. (Both of those commands are just part of the standard library; they have no particularly special place in the library and can be replaced if so desired.)

The majority of Tcl commands, especially in the standard library, are [variadic](http://en.wikipedia.org/wiki/Variadic_function" \o "Variadic function), and the proc (the constructor for scripted command procedures) supports the definition of both default values for arguments and a catch-all argument to allow the code to process arbitrary numbers of arguments.

Tcl is not statically typed: each variable may contain integers, floats, strings, lists, command names, dictionaries, or any other value; values are reinterpreted (subject to syntactic constraints) as other types on demand. However, values are [immutable](http://en.wikipedia.org/wiki/Immutable_object) and operations that appear to change them actually just return a new value instead.

## [[edit](http://en.wikipedia.org/w/index.php?title=Tcl&action=edit&section=4)]Interfacing with other languages

Tcl interfaces natively with the [C](http://en.wikipedia.org/wiki/C_(programming_language)) language. This is because it was originally written to be a framework for providing a syntactic front-end to commands written in C, and all commands in the language (including things that might otherwise be [keywords](http://en.wikipedia.org/wiki/Keyword_(computer_programming)), such as **if** or **while**) are implemented this way. Each command implementation [function](http://en.wikipedia.org/wiki/Subroutine) is passed an array of values that describe the (already substituted) arguments to the command, and is free to interpret those values as it sees fit.

Digital [logic simulators](http://en.wikipedia.org/wiki/Logic_simulation) often include a Tcl scripting interface for simulating [Verilog](http://en.wikipedia.org/wiki/Verilog" \o "Verilog), [VHDL](http://en.wikipedia.org/wiki/VHDL) and SystemVerilog [hardware languages](http://en.wikipedia.org/wiki/Hardware_description_language).

Tools exist (e.g. [SWIG](http://en.wikipedia.org/wiki/SWIG), [ffidl](http://www.elf.org/ffidl/)) to automatically generate the necessary code to connect arbitrary C functions and the Tcl runtime, and [Critcl](http://www.equi4.com/starkit/critcl.html) does the reverse, allowing embedding of arbitrary C code inside a Tcl script and compiling it at runtime into a [DLL](http://en.wikipedia.org/wiki/Dynamic-link_library).