26.5.1 Definition of `getauxval`

`unsigned long int getauxval` (*unsigned long int* `type`) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function is used to inquire about the entries in the auxiliary vector. The *type* argument should be one of the '`AT_`' symbols defined in `elf.h`. If a matching entry is found, the value is returned; if the entry is not found, zero is returned and `errno` is set to `ENOENT`.

For some platforms, the key `AT_HWCAP` is the easiest way to inquire about any instruction set extensions available at runtime. In this case, there will (of necessity) be a platform-specific set of '`HWCAP_`' values masked together that describe the capabilities of the cpu on which the program is being executed.

26.6 System Calls

A system call is a request for service that a program makes of the kernel. The service is generally something that only the kernel has the privilege to do, such as doing I/O. Programmers don't normally need to be concerned with system calls because there are functions in the GNU C Library to do virtually everything that system calls do. These functions work by making system calls themselves. For example, there is a system call that changes the permissions of a file, but you don't need to know about it because you can just use the GNU C Library's `chmod` function.

System calls are sometimes called syscalls or kernel calls, and this interface is mostly a purely mechanical translation from the kernel's ABI to the C ABI. For the set of syscalls where we do not guarantee POSIX Thread cancellation the wrappers only organize the incoming arguments from the C calling convention to the calling convention of the target kernel. For the set of syscalls where we provided POSIX Thread cancellation the wrappers set some internal state in the library to support cancellation, but this does not impact the behaviour of the syscall provided by the kernel.

The GNU C Library includes by reference the Linux man-pages 6.8 documentation to document the listed syscalls for the Linux kernel. For reference purposes only the latest Linux man-pages Project (`https://www.kernel.org/doc/man-pages/`) documentation can be accessed from the Linux kernel (`https://www.kernel.org`) website. Where the syscall has more specific documentation in this manual that more specific documentation is considered authoritative.

Here is the list of all potential non-cancellable system calls, across all configurations of the GNU C Library: `access acct adjtime alarm arch_prctl bdflush bind cachectl cacheflush capget capset chdir chflags chmod chown chroot close_range create_module delete_module dup2 dup3 dup epoll_create1 epoll_ctl eventfd execve fanotify_init fchdir fchflags fchmod fchownat fchown fgetxattr flistxattr flock fremovexattr fsconfig fsetxattr fsmount fsopen fspick fstatfs ftruncate get_kernel_syms getdents getdomain getdtsz getegid geteuid getgid getgroups gethostid gethostname getitimer getpagesize getpeername getpgid getpgrp getpid getppid getpriority getresgid getresuid getrlimit getrusage getsid getsockname getsockopt gettid getuid getxattr init_module inotify_add_watch`

`inotify_init1 inotify_rm_watch ioctl ioperm iopl killpg kill klogctl lchown lgetxattr linkat link listen listxattr llistxattr lremovexattr lseek64 lseek lsetxattr madvise memfd_create mincore mkdirat mkdir mlockall mlock mmap modify_ldt mount_setattr mount move_mount mprotect munlockall munlock munmap name_to_handle_at nfsservctl open_tree pciconfig_iobase pciconfig_read pciconfig_write personality pidfd_getfd pidfd_open pidfd_send_signal pipe2 pivot_root pkey_alloc pkey_free posix_fadvise64 prctl process_madvise process_mrelease profil ptrace query_module quotactl readlinkat readlink reboot remap_file_pages removexattr rename revoke rmdir sched_getp sched_gets sched_primax sched_primin sched_setp sched_sets sched_yield setdomain setegid seteuid setfsgid setfsuid setgid setgroups sethostid sethostname setitimer setlogin setns setpgid setpriority setregid setreuid setrlimit setsid setsockopt setuid setxattr shutdown sigaltstack sigpause sigstack socketpair socket statfs swapoff swapon symlinkat symlink syncfs sync syscall_clock_gettime sysinfo sysmips tgkill timerfd_create truncate umask uname unlinkat unlink unshare uselib utimes vhangup vm86old vm86 wait4`

Here's the corresponding list of cancellable system calls: `accept close connect fcntl open readv read recvfrom recvmsg recv select sendmsg sendto send sigsuspend writev write`

However, there are times when you want to make a system call explicitly, and for that, the GNU C Library provides the `syscall` function. `syscall` is harder to use and less portable than functions like `chmod`, but easier and more portable than coding the system call in assembler instructions.

`syscall` is most useful when you are working with a system call which is special to your system or is newer than the GNU C Library you are using. `syscall` is implemented in an entirely generic way; the function does not know anything about what a particular system call does or even if it is valid.

The description of `syscall` in this section assumes a certain protocol for system calls on the various platforms on which the GNU C Library runs. That protocol is not defined by any strong authority, but we won't describe it here either because anyone who is coding `syscall` probably won't accept anything less than kernel and C library source code as a specification of the interface between them anyway.

`syscall` is declared in `unistd.h`.

`long int syscall` (*long int sysno, ...*) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

`syscall` performs a generic system call.

sysno is the system call number. Each kind of system call is identified by a number. Macros for all the possible system call numbers are defined in `sys/syscall.h`

The remaining arguments are the arguments for the system call, in order, and their meanings depend on the kind of system call. If you code more arguments than the system call takes, the extra ones to the right are ignored.

The return value is the return value from the system call, unless the system call failed. In that case, `syscall` returns `-1` and sets `errno` to an error code that the system call returned. Note that system calls do not return `-1` when they succeed.

If you specify an invalid *sysno*, `syscall` returns `-1` with `errno` = `ENOSYS`.

Example:

```
#include <unistd.h>
#include <sys/syscall.h>
#include <errno.h>

...

int rc;

rc = syscall(SYS_chmod, "/etc/passwd", 0444);

if (rc == -1)
   fprintf(stderr, "chmod failed, errno = %d\n", errno);
```

This, if all the compatibility stars are aligned, is equivalent to the following preferable code:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>

...

int rc;

rc = chmod("/etc/passwd", 0444);
if (rc == -1)
   fprintf(stderr, "chmod failed, errno = %d\n", errno);
```

26.7 Program Termination

The usual way for a program to terminate is simply for its `main` function to return. The *exit status value* returned from the `main` function is used to report information back to the process's parent process or shell.

A program can also terminate normally by calling the `exit` function.

In addition, programs can be terminated by signals; this is discussed in more detail in Chapter 25 [Signal Handling], page 754. The `abort` function causes a signal that kills the program.

26.7.1 Normal Termination

A process terminates normally when its program signals it is done by calling `exit`. Returning from `main` is equivalent to calling `exit`, and the value that `main` returns is used as the argument to `exit`.

`void exit` (*int status*) [Function]

Preliminary: | MT-Unsafe race:exit | AS-Unsafe corrupt | AC-Unsafe corrupt lock | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `exit` function tells the system that the program is done, which causes it to terminate the process.