# UNIX — Programming and System Management I

Ondřej Kuhejda

January 21, 2020

## Contents

# 1 Foreword

Hi, fellow student! I created this in my free time. If you want to make me happy, check my dotfiles: `https://github.com/kuhy/.dotfiles`. And give me a star, plz.

# 2 Development environment

## 2.1 User commands (1)

**cpp** (*The C Preprocessor*) The C preprocessor, often known as `cpp`, is a macro processor that is used automatically by the C compiler to transform your program before compilation. It is called a macro processor because it allows you to define macros, which are brief abbreviations for longer constructs.

**as** (*The portable GNU assembler*) GNU as is really a family of assemblers. If you use (or have used) the GNU assembler on one architecture, you should find a fairly similar environment when you use it on another architecture. Each version has much in common with the others, including object file formats, most assembler directives (often called pseudo-ops) and assembler syntax.

**ld** (*The GNU linker*) `ld` combines a number of object and archive files, relocates their data and ties up symbol references. Usually the last step in compiling a program is to run `ld`.

**gcc** (*GNU project C and C++ compiler*) When you invoke GCC, it normally does preprocessing, compilation, assembly and linking. The "overall options" allow you to stop this process at an intermediate stage. For example, the `-c` option says not to run the linker. Then the output consists of object files output by the assembler.

**make** (*GNU make utility to maintain groups of programs*) The `make` utility will determine automatically which pieces of a large program need to be recompiled, and issue the commands to recompile them. The manual describes the GNU implementation of `make`, which was written by Richard Stallman and Roland McGrath, and is currently maintained by Paul Smith. Our examples show C programs, since they are very common, but you can use `make` with any programming language whose compiler can be run with a shell command. In fact, `make` is not limited to programs. You can use it to describe any task where some files must be updated automatically from others whenever the others change.

**nm** (*list symbols from object files*) GNU `nm` lists the symbols from object files. If no object files are listed as arguments, `nm` assumes the file `a.out`.

**strip** (*discard symbols from object files*) GNU `strip` discards all symbols from object files. The list of object files may include archives. At least one object file must be given.

**size** (*list section sizes and total size*) The GNU size utility lists the section sizes and the total size for each of the object or archive files objfile in its argument list. By default, one line of output is generated for each object file or each module in an archive.

**objdump** (*display information from object files*) `objdump` displays information about one or more object files. The options control what particular information to display. This information is mostly useful to programmers who are working on the compilation tools, as opposed to programmers who just want their program to compile and work.

**strings** (*print the strings of printable characters in files*) For each file given, GNU `strings` prints the printable character sequences that are at least 4 characters long (or the number given with the options below) and are followed by an unprintable character. strings is mainly useful for determining the contents of non-text files.

**ar** (*create, modify, and extract from archives*) The GNU `ar` program creates, modifies, and extracts from archives. An archive is a single file holding a collection of other files in a structure that makes it possible to retrieve the original individual files (called members of the archive).

**ranlib** (*generate index to archive*) `ranlib` generates an index to the contents of an archive and stores it in the archive. The index lists each symbol defined by a member of an archive that is a relocatable object file. You may use `nm -s` or `nm --print-armap` to list this index. An archive with such an index speeds up linking to the library and allows routines in the library to call each other without regard to their placement in the archive. The GNU `ranlib` program is another form of GNU `ar`; running `ranlib` is completely equivalent to executing `ar -s`.

**ldd** (*print shared object dependencies*) `ldd` prints the shared objects (shared libraries) required by each program or shared object specified on the command line.

**ulimit** (*set or get resource usage limits*) `ulimit` builtin sets or outputs the resource usage limits of the shell and any processes spawned by it. If a new limit value is omitted, the current value of the limit of the resource is printed; otherwise, the specified limit is set to the new value.

**coredumpctl** (*Retrieve and process saved core dumps and metadata*) `coredumpctl` is a tool that can be used to retrieve and process core dumps and metadata which were saved by `systemd-coredump(8)`.

**gdb** (*The GNU Debugger*) The purpose of a debugger such as `GDB` is to allow you to see what is going on "inside" another program while it executes – or what another program was doing at the moment it crashed.

**dbx** (*source-level debugging tool*) `dbx` is a utility for source-level debugging and execution of programs written in C++, ANSI C, Fortran 77, Fortran 95, and Java programming languages.

**xxgdb** (*X window system interface to the gdb debugger*) Xxgdb is a graphical user interface to the `gdb` debugger under the X Window System. It provides visual feedback and mouse input for the user to control program execution through breakpoints, to examine and traverse the function call stack, to display values of variables and data structures, and to browse source files and functions.

**ddd** (*The Data Display Debugger*) DDD is a graphical front-end for GDB and other command-line debuggers.

## 2.2 System calls (2)

**ptrace** (*process trace*) The `ptrace()` system call provides a means by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's memory and registers. It is primarily used to implement breakpoint debugging and system call tracing.

```
long ptrace(enum __ptrace_request request, pid_t pid,
    void *addr, void *data);
```

## 2.3 File formats and conventions (5)

**elf** (*format of Executable and Linking Format files*) The header file `<elf.h>` defines the format of ELF executable binary files. Amongst these files are normal executable files, relocatable object files, core files, and shared objects. An executable file using the ELF file format consists of an ELF header, followed by a program header table or a section header table, or both. The ELF header is always at offset zero of the file. The program header table and the section header table's offset in the file are defined in the ELF header. The two tables describe the rest of the particularities of the file. This header file describes the above mentioned headers as C structures and also includes structures for dynamic sections, relocation sections and symbol tables.

**a.out** (*format of executable binary files*) The include file `<a.out.h>` declares three structures and several macros. The structures describe the format of executable machine code files ('binaries') on the system.

**core** (*core dump file*) The default action of certain signals is to cause a process to terminate and produce a core dump file, a disk file containing an image of the process's memory at the time of termination. This image can be used in a debugger (e.g., `gdb(1)`) to inspect the state of the program at the time that it terminated. A list of the signals which cause a process to dump core can be found in `signal(7)`. A process can set its soft `RLIMIT_CORE` resource limit to place an upper limit on the size of the core dump file that will be produced if it receives a "core dump" signal; see `getrlimit(2)` for details.

**proc** (*process information pseudo-filesystem*) The `proc` filesystem is a pseudo-filesystem which provides an interface to kernel data structures. It is commonly mounted at `/proc`. Most of the files in the proc filesystem are read-only, but some files are writable, allowing kernel variables to be changed.

## 2.4 System management commands (8)

**ld.so** (*dynamic linker/loader*) The programs `ld.so` and `ld-linux.so*` find and load the shared objects (shared libraries) needed by a program, prepare the program to run, and then run it. Linux binaries require dynamic linking (linking at run time) unless the `-static` option was given to `ld(1)` during compilation. The program `ld.so` handles `a.out` binaries, a format used long ago; `ld-linux.so*` handles ELF, which everybody has been using for years now.

**Environment variables** `LD_LIBRARY_PATH` A list of directories in which to search for ELF libraries at execution time. The items in the list are separated by either colons or semicolons, and there is no support for escaping either separator.

`LD_PRELOAD` A list of additional, user-specified, ELF shared objects to be loaded before all others.

**ldconfig** (*configure dynamic linker run-time bindings*) `ldconfig` creates the necessary links and cache to the most recent shared libraries found in the directories specified on the command line, in the file `/etc/ld.so.conf`, and in the trusted directories, `/lib` and `/usr/lib` (on some 64-bit architectures such as x86-64, `/lib` and `/usr/lib` are the trusted directories for 32-bit libraries, while `/lib64` and `/usr/lib64` are used for 64-bit libraries). The cache is used by the run-time linker, `ld.so` or `ld-linux.so`. `ldconfig` checks the header and filenames of the libraries it encounters when determining which versions should have their links updated.

**prelink** (*prelink ELF shared libraries and binaries to speed up startup time*) `prelink` is a program that modifies ELF shared libraries and ELF dynamically linked binaries in such a way that the time needed for the dynamic linker to perform relocations at startup significantly decreases. Due to fewer relocations, the run-time memory consumption decreases as well (especially

the number of unshareable pages). The prelinking information is only used at startup time if none of the dependent libraries have changed since prelinking; otherwise programs are relocated normally.

**systemd-coredump** (*Acquire, save and process core dumps*) systemd coredump service is a system service that can acquire core dumps from the kernel and handle them in various ways. The `systemd-coredump` executable does the actual work. It is invoked twice: once as the handler by the kernel, and the second time in the systemd coredump service to actually write the data to the journal.

# 3 API standards

## 3.1 Header files (0)

**limits.h** (*implementation-defined constants*) The `<limits.h>` header shall define macros and symbolic constants for various limits.

**float.h** (*floating types*) Contains a set of various platform-dependent constants related to floating point values.

**stdio.h** (*standard buffered input/output*) The `stdio.h` header defines three variable types, several macros, and various functions for performing input and output.

## 3.2 Library calls (3)

**sysconf** (*get configuration information at run time*) POSIX allows an application to test at compile or run time whether certain options are supported, or what the value is of certain configurable constants or limits. At compile time this is done by including `<unistd.h>` and/or `<limits.h>` and testing the value of certain macros. At run time, one can ask for numerical values using the present function `sysconf()`. One can ask for numerical values that may depend on the filesystem in which a file resides using `fpathconf(3)` and `pathconf(3)`. One can ask for string values using `confstr(3)`.

```
long sysconf(int name);
```

**pathconf** (*get configuration values for files*) `pathconf()` gets a value for configuration option name for the filename path.

```
long pathconf(const char *path, int name);
```

**fpathconf** (*get configuration values for files*) `fpathconf()` gets a value for the configuration option name for the open file descriptor `fd`.

```
long fpathconf(int fd, int name);
```

## 3.3 Miscellaneous (7)

**standards** (*C and UNIX Standards*) The `CONFORMING TO` section that appears in many manual pages identifies various standards to which the documented interface conforms. The following list briefly describes these standards.

**C89** This was the first C language standard, ratified by ANSI (American National Standards Institute) in 1989 (X3.159-1989). Sometimes this is known as ANSI C, but since C99 is also an ANSI standard, this term is ambiguous. This standard was also ratified by ISO (International Standards Organization) in 1990 (ISO/IEC 9899:1990), and is thus occasionally referred to as ISO C90.

**POSIX.1-1990** "Portable Operating System Interface for Computing Environments". IEEE 1003.1-1990 part 1, ratified by ISO in 1990 (ISO/IEC 9945-1:1990). The term "POSIX" was coined by Richard Stallman.

**SUSv3** This was a 2001 revision and consolidation of the POSIX.1, POSIX.2, and SUS standards into a single document, conducted under the auspices of the Austin Group (`http://www.opengroup.org/austin/`). The standard is available online at (`http://www.unix-systems.org/version3/`), and the interfaces that it describes are also available in the Linux manual pages package under sections 1p and 3p (e.g., "man 3p open").

**XPG3** Released in 1989, this was the first significant release of the X/Open Portability Guide, produced by the X/Open Company, a multivendor consortium. This multivolume guide was based on the POSIX standards.

**SVID 4** System V Interface Definition version 4, issued in 1995. Available online at (`http://www.sco.com/developers/devspecs/`).

**4.2BSD** This is an implementation standard defined by the 4.2 release of the Berkeley Software Distribution, released by the University of California at Berkeley. This was the first Berkeley release that contained a TCP/IP stack and the sockets API. 4.2BSD was released in 1983.

# 4 User-space programs

## 4.1 User commands (1)

**valgrind** (*a suite of tools for debugging and profiling programs*) Valgrind is a flexible program for debugging and profiling Linux executables. It consists of a core, which provides a synthetic CPU in software, and a series of debugging and profiling tools. The architecture is modular, so that new tools can be created easily and without disturbing the existing structure.

**gettext** (*translate message*) The `gettext` program translates a natural language message into the user's language, by looking up the translation in a message catalog. Display native language translation of a textual message.

**locale** (*get locale-specific information*) The `locale` command displays information about the current locale, or all locales, on standard output.

**localedef** (*compile locale definition files*) The `localedef` program reads the indicated charmap and input files, compiles them to a binary form quickly usable by the locale functions in the C library (`setlocale(3)`, `localeconv(3)`, etc.), and places the output in outputpath.

## 4.2 System calls (2)

**_exit** (*terminate the calling process*) The function `_exit()` terminates the calling process "immediately". Any open file descriptors belonging to the process are closed. Any children of the process are inherited by `init(1)` (or by the nearest "subreaper" process as defined through the use of the `prctl(2)` `PR_SET_CHILD_SUBREAPER` operation). The process's parent is sent a `SIGCHLD` signal. The value `status & 0377` is returned to the parent process as the process's exit status, and can be collected using one of the `wait(2)` family of calls. The function `_Exit()` is equivalent to `_exit()`.

```
void _exit(int status);
```

**brk** (*change data segment size*) `brk()` and `sbrk()` change the location of the program break, which defines the end of the process's data segment (i.e., the program break is the first location after the end of the uninitialized data segment). Increasing the program break has the effect of allocating memory to the process; decreasing the break deallocates memory. `brk()` sets the end of the data segment to the value specified by `addr`, when that value is reasonable,

the system has enough memory, and the process does not exceed its maximum data size (see `setrlimit(2)`).

```
int brk(void *addr);
```

**sbrk** (*change data segment size*) `sbrk()` increments the program's data space by increment bytes. Calling `sbrk()` with an increment of 0 can be used to find the current location of the program break.

```
void *sbrk(intptr_t increment);
```

## 4.3 Library calls (3)

**exit** (*cause normal process termination*) The `exit()` function causes normal process termination and the value of status `& 0377` is returned to the parent (see `wait(2)`). All functions registered with `atexit(3)` and `on_exit(3)` are called, in the reverse order of their registration. All open `stdio(3)` streams are flushed and closed. Files created by `tmpfile(3)` are removed. The C standard specifies two constants, `EXIT_SUCCESS` and `EXIT_FAILURE`, that may be passed to `exit()` to indicate successful or unsuccessful termination, respectively.

```
void exit(int status);
```

**atexit** (*register a function to be called at normal process termination*) The `atexit()` function registers the given function to be called at normal process termination, either via `exit(3)` or via return from the program's `main()`. Functions so registered are called in the reverse order of their registration; no arguments are passed. The same function may be registered multiple times: it is called once for each registration.

```
int atexit(void (*function)(void));
```

**abort** (*cause abnormal process termination*) The `abort()` first unblocks the `SIGABRT` signal, and then raises that signal for the calling process (as though `raise(3)` was called). This results in the abnormal termination of the process unless the `SIGABRT` signal is caught and the signal handler does not return (see `longjmp(3)`).

```
void abort(void);
```

**getopt** (*parse command options*) The `getopt()` function parses the command-line arguments. Its arguments `argc` and `argv` are the argument count and array as passed to the `main()` function on program invocation. An element of `argv` that starts with `-` (and is not exactly `-` or `--`) is an option element. The characters of this element (aside from the initial `-`) are option characters. If `getopt()` is called repeatedly, it returns successively each of the option characters from each of the option elements.

```
int getopt(int argc, char * const argv[],
   const char *optstring);
```

**getopt_long** (*parse command options*) The `getopt_long()` function works like `getopt()` except that it also accepts long options, started with two dashes.

```
int getopt_long(int argc, char * const argv[],
   const char *optstring,
   const struct option *longopts, int *longindex);
```

**errno** (*number of last error*) The `<errno.h>` header file defines the integer variable `errno`, which is set by system calls and some library functions in the event of an error to indicate what went wrong.

**perror** (*print a system error message*) The `perror()` function produces a message on standard error describing the last error encountered during a call to a system or library function. When a system call fails, it usually returns `-1` and sets the variable `errno` to a value describing what went wrong. (These values can be found in `<errno.h>`.) Many library functions do likewise. The function `perror()` serves to translate this error code into human-readable form.

```
void perror(const char *s);
```

**strerror** (*return string describing error number*) The `strerror()` function returns a pointer to a string that describes the error code passed in the argument errnum, possibly using the `LC_MESSAGES` part of the current locale to select the appropriate language. (For example, if `errnum` is `EINVAL`, the returned description will be "Invalid argument".)

```
char *strerror(int errnum);
```

**getenv** (*get an environment variable*) The `getenv()` function searches the environment list to find the environment variable name, and returns a pointer to the corresponding value string.

```
char *getenv(const char *name);
```

**putenv** (*change or add an environment variable*) The `putenv()` function adds or changes the value of environment variables. The argument string is of the form `name=value`. If name does not already exist in the environment, then string is added to the environment. If name does exist, then the value of name in the environment is changed to value. The string pointed to by string becomes part of the environment, so altering the string changes the environment.

```
int putenv(char *string);
```

**setenv** (*change or add an environment variable*) The `setenv()` function adds the variable name to the environment with the value `value`, if name does not already exist. If name does exist in the environment, then its value is changed to `value` if overwrite is nonzero; if overwrite is zero, then the value of name is not changed (and `setenv()` returns a success status). This function makes copies of the strings pointed to by name and value (by contrast with `putenv(3)`).

```
int setenv(const char *name, const char *value, int overwrite);
```

**unsetenv** (*delete an environment variable*) The `unsetenv()` function deletes the variable name from the environment. If name does not exist in the environment, then the function succeeds, and the environment is unchanged.

```
int unsetenv(const char *name);
```

**clearenv** (*clear the environment*) The `clearenv()` function clears the environment of all name-value pairs and sets the value of the external variable environ to `NULL`. After this call, new variables can be added to the environment using `putenv(3)` and `setenv(3)`.

```
int clearenv(void);
```

**malloc** (*allocate dynamic memory*) The `malloc()` function allocates size bytes and returns a pointer to the allocated memory. The memory is not initialized. If size is `0`, then `malloc()` returns either `NULL`, or a unique pointer value that can later be successfully passed to `free()`.

```
void *malloc(size_t size);
```

**calloc** (*allocate dynamic memory*) The `calloc()` function allocates memory for an array of nmemb elements of size bytes each and returns a pointer to the allocated memory. The memory is set to zero. If `nmemb` or `size` is `0`, then `calloc()` returns either `NULL`, or a unique pointer value that can later be successfully passed to `free()`.

```
void *calloc(size_t nmemb, size_t size);
```

**realloc** (*reallocate dynamic memory*) The `realloc()` function changes the size of the memory block pointed to by `ptr` to size bytes. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. If the new size is larger than the old size, the added memory will not be initialized. If `ptr` is `NULL`, then the call is equivalent to malloc(size), for all values of size; if size is equal to zero, and `ptr` is not `NULL`, then the call is equivalent to `free(ptr)`.

```
void *realloc(void *ptr, size_t size);
```

**free** (*free dynamic memory*) The `free()` function frees the memory space pointed to by `ptr`, which must have been return a previous call to `malloc()`, `calloc()`, or `realloc()`. Otherwise, or if `free(ptr)` has already been called before, undefined behavior occurs. If `ptr` is `NULL`, no operation is performed.

```
void *calloc(size_t nmemb, size_t size);
```

**alloca** (*allocate memory that is automatically freed*) The `alloca()` function allocates `size` bytes of space in the stack frame of the caller. This temporary space is automatically freed when the function that called `alloca()` returns to its caller.

```
void *alloca(size_t size);
```

**efence** (*Electric Fence Malloc Debugger*) Electric Fence helps you detect two common programming bugs: software that overruns the boundaries of a `malloc()` memory allocation, and software that touches a memory allocation that has been released by `free()`. Unlike other `malloc()` debuggers, Electric Fence will detect read accesses as well as writes, and it will pinpoint the exact instruction that causes an error.

**setjmp** (*performing a nonlocal goto*) The `setjmp()` function saves various information about the calling environment (typically, the stack pointer, the instruction pointer, possibly the values of other registers and the signal mask) in the buffer `env` for later use by `longjmp()`. In this case, `setjmp()` returns `0`.

```
int setjmp(jmp_buf env);
```

**longjmp** (*performing a nonlocal goto*) The `longjmp()` function uses the information saved in env to transfer control back to the point where `setjmp()` was called and to restore ("rewind") the stack to its state at the time of the `setjmp()` call. In addition, and depending on the implementation, the values of some other registers and the process signal mask may be restored to their state at the time of the `setjmp()` call. Following a successful `longjmp()`, execution continues as if `setjmp()` had returned for a second time. This "fake" return can be distinguished from a true `setjmp()` call because the "fake" return returns the value provided in val. If the programmer mistakenly passes the value `0` in val, the "fake" return will instead return `1`.

```
void longjmp(jmp_buf env, int val);
```

**dlopen** (*open a shared object*) The function `dlopen()` loads the dynamic shared object (shared library) file named by the null-terminated string filename and returns an opaque "handle" for the loaded object. This handle is employed with other functions in the `dlopen` API, such as `dlsym(3)`, `dladdr(3)`, `dlinfo(3)`, and `dlclose()`.

```
void *dlopen(const char *filename, int flags);
```

One of the following two values must be included in flags:

`RTLD_LAZY` Perform lazy binding. Resolve symbols only as the code that references them is executed. If the symbol is never referenced, then it is never resolved. (Lazy binding is performed only for function references; references to variables are always immediately bound when the shared object is loaded.) Since `glibc` 2.1.1, this flag is overridden by the effect of the `LD_BIND_NOW` environment variable.

`RTLD_NOW` If this value is specified, or the environment variable `LD_BIND_NOW` is set to a nonempty string, all undefined symbols in the shared object are resolved before `dlopen()` returns. If this cannot be done, an error is returned.

Zero or more of the following values may also be ORed in flags:

`RTLD_GLOBAL` The symbols defined by this shared object will be made available for symbol resolution of subsequently loaded shared objects.

**dlclose** (*close a shared object*) The function `dlclose()` decrements the reference count on the dynamically loaded shared object referred to by handle. If the reference count drops to zero, then the object is unloaded. All shared objects that were automatically loaded when `dlopen()` was invoked on the object referred to by handle are recursively closed in the same manner.

```
int dlclose(void *handle);
```

**dlsym** (*obtain address of a symbol in a shared object or executable*) The function `dlsym()` takes a "handle" of a dynamic loaded shared object returned by `dlopen(3)` along with a null-terminated symbol name, and returns the address where that symbol is loaded into memory. If the symbol is not found, in the specified object or any of the shared objects that were automatically loaded by `dlopen(3)` when that object was loaded, `dlsym()` returns `NULL`. (The search performed by `dlsym()` is breadth first through the dependency tree of these shared objects.)

```
void *dlsym(void *handle, const char *symbol);
```

**dlerror** (*obtain error diagnostic for functions in the dlopen API*) The `dlerror()` function returns a human-readable, null-terminated string describing the most recent error that occurred from a call to one of the functions in the dlopen API since the last call to `dlerror()`. The returned string does not include a trailing newline.

```
char *dlerror(void);
```

**setlocale** (*set the current locale*) The `setlocale()` function is used to set or query the program's current locale. If `locale` is not `NULL`, the program's current locale is modified according to the arguments. The argument category determines which parts of the program's current locale should be modified.

```
char *setlocale(int category, const char *locale);
```

| Category | Governs |
|---|---|
| `LC_ALL` | All of the locale |
| `LC_COLLATE` | String collation |
| `LC_CTYPE` | Character classification |
| `LC_MESSAGES` | Localizable natural-language messages |
| `LC_MONETARY` | Formatting of monetary values |
| `LC_NUMERIC` | Formatting of nonmonetary numeric values |
| `LC_TIME` | Formatting of date and time values |

If locale is an empty string, "", each part of the locale that should be modified is set according to the environment variables.

If locale is NULL, the current locale is only queried, not modified.

A locale name is typically of the form language[_territory][.codeset][@modifier], where language is an ISO 639 language code, territory is an ISO 3166 country code, and codeset is a character set or encoding identifier like ISO-8859-1 or UTF-8. For a list of all supported locales, try locale -a (see locale(1)).

On startup of the main program, the portable "C" locale is selected as default. A program may be made portable to all locales by calling:

```
setlocale(LC_ALL, "");
```

**strcoll** (*compare two strings using the current locale*) The strcoll() function compares the two strings s1 and s2. It returns an integer less than, equal to, or greater than zero if s1 is found, respectively, to be less than, to match, or be greater than s2. The comparison is based on strings interpreted as appropriate for the program's current locale for category LC_COLLATE. (See setlocale(3).)

```
int strcoll(const char *s1, const char *s2);
```

**strxfrm** (*string transformation*) The strxfrm() function transforms the src string into a form such that the result of strcmp(3) on two strings that have been transformed with strxfrm() is the same as the result of strcoll(3) on the two strings before their transformation. The first n bytes of the transformed string are placed in dest. The transformation is based on the program's current locale for category LC_COLLATE. (See setlocale(3)).

```
size_t strxfrm(char *dest, const char *src, size_t n);
```

**nl_langinfo** (*query language and locale information*) The nl_langinfo() and nl_langinfo_l() functions provide access to locale information in a more flexible way than localeconv(3). nl_langinfo() returns a string which is the value corresponding to item in the program's current global locale. nl_langinfo() returns a string which is the value corresponding to item for the locale identified by the locale object locale, which was previously created by newlocale(1). Individual and additional elements of the locale categories can be queried.

```
char *nl_langinfo(nl_item item);

char *nl_langinfo_l(nl_item item, locale_t locale);
```

Examples for the locale elements that can be specified in item using the constants defined in <langinfo.h> are:

CODESET (LC_CTYPE) Return a string with the name of the character encoding used in the selected locale, such as "UTF-8", "ISO-8859-1", or "ANSI_X3.4-1968" (better known as US-ASCII). This is the same string that you get with "locale charmap". For a list of character encoding names, try locale -m (see locale(1)).

D_T_FMT (LC_TIME) Return a string that can be used as a format string for strftime(3) to represent time and date in a locale-specific way.

D_FMT (LC_TIME) Return a string that can be used as a format string for strftime(3) to represent a date in a locale-specific way.

T_FMT (LC_TIME) Return a string that can be used as a format string for strftime(3) to represent a time in a locale-specific way.

DAY_{1-7} (LC_TIME) Return name of the n-th day of the week. (Warning: this follows the US convention DAY_1 = Sunday, not the international convention ISO 8601 that Monday is the first day of the week.)

`ABDAY_{1-7}` (`LC_TIME`) Return abbreviated name of the n-th day of the week.

`MON_{1-12}` (`LC_TIME`) Return name of the n-th month.

`ABMON_{1-12}` (`LC_TIME`) Return abbreviated name of the n-th month.

`RADIXCHAR` (`LC_NUMERIC`) Return radix character (decimal dot, decimal comma, etc.).

**iconv** (*perform character set conversion*) The `iconv()` function converts a sequence of characters in one character encoding to a sequence of characters in another character encoding. The `cd` argument is a conversion descriptor, previously created by a call to `iconv_open(3)`; the conversion descriptor defines the character encodings that `iconv()` uses for the conversion.

```
size_t iconv(iconv_t cd,
     char **inbuf, size_t *inbytesleft,
     char **outbuf, size_t *outbytesleft);
```

**iconv_open** (*allocate descriptor for character set conversion*) The `iconv_open()` function allocates a conversion descriptor suitable for converting byte sequences from character encoding `fromcode` to character encoding `tocode`.

```
iconv_t iconv_open(const char *tocode, const char *fromcode);
```

**iconv_close** (*deallocate descriptor for character set conversion*) The `iconv_close()` function deallocates a conversion descriptor `cd` previously allocated using `iconv_open(3)`.

```
int iconv_close(iconv_t cd);
```

### 4.4 Miscellaneous (7)

**libc** (*overview of standard C libraries on Linux*) The term "libc" is commonly used as a shorthand for the "standard C library", a library of standard functions that can be used by all C programs (and sometimes by programs in other languages). The pathname `/lib/libc.so.6` (or something similar) is normally a symbolic link that points to the location of the glibc library, and executing this pathname will cause glibc to display various information about the version installed on your system.

**environ** (*user environment*) The variable environ points to an array of pointers to strings called the "environment". The last pointer in this array has the value `NULL`. This array of strings is made available to the process by the `exec(3)` call that started the process. When a child process is created via `fork(2)`, it inherits a copy of its parent's environment.

## 5 Kernel

### 5.1 User commands (1)

**systemd** (*systemd system and service manager*) systemd is a system and service manager for Linux operating systems. When run as first process on boot (as PID 1), it acts as init system that brings up and maintains userspace services.

### 5.2 System calls (2)

**idle** (*make process 0 idle*) `idle()` is an internal system call used during bootstrap. It marks the process's pages as swappable, lowers its priority, and enters the main scheduling loop. `idle()` never returns.

Only process 0 may call `idle()`. Any user process, even a process with superuser permission, will receive EPERM.

```
int idle(void);
```

## 5.3 Special files (4)

**initrd** boot loader initialized RAM disk The special file /dev/initrd is a read-only block device. This device is a RAM disk that is initialized (e.g., loaded) by the boot loader before the kernel is started. The kernel then can use /dev/initrd's contents for a two-phase system boot-up.

In the first boot-up phase, the kernel starts up and mounts an initial root filesystem from the contents of /dev/initrd (e.g., RAM disk initialized by the boot loader). In the second phase, additional drivers or other modules are loaded from the initial root device's contents. After loading the additional modules, a new root filesystem (i.e., the normal root filesystem) is mounted from a different device.

## 5.4 File formats and conventions (5)

**slabinfo** (*kernel slab allocator statistics*) Frequently used objects in the Linux kernel (buffer heads, inodes, dentries, etc.) have their own cache. The file /proc/slabinfo gives statistics on these caches.

## 5.5 Miscellaneous (7)

**boot** (*System bootup process based on UNIX System V Release 4*) The bootup process (or "boot sequence") varies in details among systems, but can be roughly divided into phases controlled by the following components:

1. hardware
2. operating system (OS) loader
3. kernel
4. root user-space process (init and inittab)
5. boot scripts

Each of these is described below in more detail.

**Hardware** After power-on or hard reset, control is given to a program stored in read-only memory (normally PROM); for historical reasons involving the personal computer, this program is often called "the BIOS".

This program normally performs a basic self-test of the machine and accesses nonvolatile memory to read further parameters. This memory in the PC is battery-backed CMOS memory, so most people refer to it as "the CMOS"; outside of the PC world, it is usually called "the NVRAM" (nonvolatile RAM).

The parameters stored in the NVRAM vary among systems, but as a minimum, they should specify which device can supply an OS loader, or at least which devices may be probed for one; such a device is known as "the boot device". The hardware boot stage loads the OS loader from a fixed position on the boot device, and then transfers control to it.

`Note`: The device from which the OS loader is read may be attached via a network, in which case the details of booting are further specified by protocols such as DHCP, TFTP, PXE, Etherboot, etc.

**OS loader** The main job of the OS loader is to locate the kernel on some device, load it, and run it. Most OS loaders allow interactive use, in order to enable specification of an alternative kernel (maybe a backup in case the one last compiled isn't functioning) and to pass optional parameters to the kernel.

In a traditional PC, the OS loader is located in the initial 512-byte block of the boot device; this block is known as "the MBR" (Master Boot Record).

In most systems, the OS loader is very limited due to various constraints. Even on non-PC systems, there are some limitations on the size and complexity of this loader,

but the size limitation of the PC MBR (512 bytes, including the partition table) makes it almost impossible to squeeze much functionality into it.

Therefore, most systems split the role of loading the OS between a primary OS loader and a secondary OS loader; this secondary OS loader may be located within a larger portion of persistent storage, such as a disk partition.

In Linux, the OS loader is often either `lilo(8)` or `grub(8)`.

**Kernel** When the kernel is loaded, it initializes various components of the computer and operating system; each portion of software responsible for such a task is usually consider "a driver" for the applicable component. The kernel starts the virtual memory swapper (it is a kernel process, called "kswapd" in a modern Linux kernel), and mounts some filesystem at the root path, `/`.

Some of the parameters that may be passed to the kernel relate to these activities (for example, the default root filesystem can be overridden); for further information on Linux kernel parameters, read `bootparam(7)`.

Only then does the kernel create the initial userland process, which is given the number 1 as its PID (process ID). Traditionally, this process executes the program `/sbin/init`, to which are passed the parameters that haven't already been handled by the kernel.

**bootparam** (*introduction to boot time parameters of the Linux kernel*) The Linux kernel accepts certain "command-line options" or "boot time parameters" at the moment it is started. In general, this is used to supply the kernel with information about hardware parameters that the kernel would not be able to determine on its own, or to avoid/override the values that the kernel would otherwise detect.

When the kernel is booted directly by the BIOS, you have no opportunity to specify any parameters. So, in order to take advantage of this possibility you have to use a boot loader that is able to pass parameters, such as GRUB.

**numa** (*overview of Non-Uniform Memory Architecture*) Non-Uniform Memory Access (NUMA) refers to multiprocessor systems whose memory is divided into multiple memory nodes. The access time of a memory node depends on the relative locations of the accessing CPU and the accessed node. (This contrasts with a symmetric multiprocessor system, where the access time for all of the memory is the same for all CPUs.) Normally, each CPU on a NUMA system has a local memory node whose contents can be accessed faster than the memory in the node local to another CPU or the memory on a bus shared by all CPUs.

### 5.6 System management commands (8)

**dmesg** (*print or control the kernel ring buffer*) dmesg is used to examine or control the kernel ring buffer. The default action is to display all messages from the kernel ring buffer.

**depmod** (*generate modules.dep and map files*) Linux kernel modules can provide services (called "symbols") for other modules to use (using one of the `EXPORT_SYMBOL` variants in the code). If a second module uses this symbol, that second module clearly depends on the first module. These dependencies can get quite complex.

depmod creates a list of module dependencies by reading each module under `/lib/modules/version` and determining what symbols it exports and what symbols it needs.

## 6 Processes

### 6.1 User commands (1)

**ps** (*report a snapshot of the current processes.*) ps displays information about a selection of the active processes. If you want a repetitive update of the selection and the displayed information, use `top(1)` instead.

**file** (*determine file type*) file tests each argument in an attempt to classify it. There are three sets of tests, performed in this order: filesystem tests, magic tests, and language tests. The first test that succeeds causes the file type to be printed.

**newgrp** (*log in to a new group*) The newgrp command is used to change the current group ID during a login session. If the optional - flag is given, the user's environment will be reinitialized as though the user had logged in, otherwise the current environment, including current working directory, remains unchanged.

newgrp changes the current real group ID to the named group, or to the default group listed in /etc/passwd if no group name is given.

**renice** (*alter priority of running processes*) renice alters the scheduling priority of one or more running processes. The first argument is the priority value to be used. The other arguments are interpreted as process IDs (by default), process group IDs, user IDs, or user names. renice'ing a process group causes all processes in the process group to have their scheduling priority altered. renice'ing a user causes all processes owned by the user to have their scheduling priority altered.

## 6.2 System calls (2)

**syscalls** (*Linux system calls*) The system call is the fundamental interface between an application and the Linux kernel. System calls are generally not invoked directly, but rather via wrapper functions in glibc (or perhaps some other library). For details of direct invocation of a system call, see `intro(2)`. Often, but not always, the name of the wrapper function is the same as the name of the system call that it invokes. For example, glibc contains a function `truncate()` which invokes the underlying "truncate" system call.

**fork** (*create a child process*) `fork()` creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process.

The child process and the parent process run in separate memory spaces. At the time of `fork()` both memory spaces have the same content. Memory writes, file mappings (`mmap(2)`), and unmappings (`munmap(2)`) performed by one of the processes do not affect the other.

The child process is an exact duplicate of the parent process except for the following points:

- The child has its own unique process ID, and this PID does not match the ID of any existing process group (`setpgid(2)`) or session.
- The child's parent process ID is the same as the parent's process ID.
- The child does not inherit its parent's memory locks (mlock(2), `mlockall(2)`).
- Process resource utilizations (`getrusage(2)`) and CPU time counters (`times(2)`) are reset to zero in the child.
- The child's set of pending signals is initially empty (`sigpending(2)`).
- The child does not inherit semaphore adjustments from its parent (`semop(2)`).
- The child does not inherit process-associated record locks from its parent (`fcntl(2)`). (On the other hand, it does inherit `fcntl(2)` open file description locks and `flock(2)` locks from its parent.)
- The child does not inherit timers from its parent (`setitimer(2)`, `alarm(2)`, `timer_create(2)`).
- The child does not inherit outstanding asynchronous I/O operations from its parent (`aio_read(3)`, `aio_write(3)`), nor does it inherit any asynchronous I/O contexts from its parent (see `io_setup(2)`).

```
pid_t fork(void);
```

**vfork** (*create a child process and block parent*) `vfork()`, just like `fork(2)`, creates a child process of the calling process. For details and return value and errors, see `fork(2)`.

`vfork()` is a special case of `clone(2)`. It is used to create new processes without copying the page tables of the parent process. It may be useful in performance-sensitive applications where a child is created which then immediately issues an `execve(2)`.

`vfork()` differs from `fork(2)` in that the calling thread is suspended until the child terminates (either normally, by calling `_exit(2)`, or abnormally, after delivery of a fatal signal), or it makes a call to `execve(2)`. Until that point, the child shares all memory with its parent, including the stack. The child must not return from the current function or call `exit(3)` (which would have the effect of calling exit handlers established by the parent process and flushing the parent's `stdio(3)` buffers), but may call `_exit(2)`.

```
pid_t vfork(void);
```

**wait** (*wait for process to change state*) All of these system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal. In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state.

The `wait()` system call suspends execution of the calling thread until one of its children terminates. The call `wait(&wstatus)` is equivalent to:

```
waitpid(-1, &wstatus, 0);
```

**waitpid** (*wait for process to change state*) The `waitpid()` system call suspends execution of the calling thread until a child specified by pid argument has changed state. By default, `waitpid()` waits only for terminated children, but this behavior is modifiable via the options argument, as described below.

```
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

The value of pid can be:

`< -1` meaning wait for any child process whose process group ID is equal to the absolute value of pid.

`-1` meaning wait for any child process.

`0` meaning wait for any child process whose process group ID is equal to that of the calling process.

`> 0` meaning wait for the child whose process ID is equal to the value of pid.

The value of options is an OR of zero or more of the following constants:

`WNOHANG` return immediately if no child has exited.

`WUNTRACED` also return if a child has stopped (but not traced via `ptrace(2)`). Status for traced children which have stopped is provided even if this option is not specified.

`WCONTINUED` also return if a stopped child has been resumed by delivery of `SIGCONT`.

If wstatus is not NULL, `wait()` and `waitpid()` store status information in the int to which it points. This integer can be inspected with the following macros (which take the integer itself as an argument, not a pointer to it, as is done in `wait()` and `waitpid()`!):

`WIFEXITED(wstatus)` returns true if the child terminated normally, that is, by calling `exit(3)` or `_exit(2)`, or by returning from `main()`.

`WIFSIGNALED(wstatus)` returns true if the child process was terminated by a signal.

`WCOREDUMP(wstatus)` returns true if the child produced a core dump. This macro should be employed only if `WIFSIGNALED` returned true.

This macro is not specified in POSIX.1-2001 and is not available on some UNIX implementations (e.g., AIX, SunOS).

`WIFSTOPPED(wstatus)` returns true if the child process was stopped by delivery of a signal; this is possible only if the call was done using `WUNTRACED` or when the child is being traced (see `ptrace(2)`).

**wait3, wait4** (*wait for process to change state, BSD style*) These functions are nonstandard; in new programs, the use of `waitpid(2)` or `waitid(2)` is preferable.

The `wait3()` and `wait4()` system calls are similar to `waitpid(2)`, but additionally return resource usage information about the child in the structure pointed to by rusage.

`wait3()` waits of any child, while `wait4()` can be used to select a specific child, or children, on which to wait.

```
pid_t wait3(int *wstatus, int options,
    struct rusage *rusage);

pid_t wait4(pid_t pid, int *wstatus, int options,
    struct rusage *rusage);
```

**execve** (*execute program*) `execve()` executes the program pointed to by filename. This causes the program that is currently being run by the calling process to be replaced with a new program, with newly initialized stack, heap, and (initialized and uninitialized) data segments.

```
int execve(const char *filename, char *const argv[],
   char *const envp[]);
```

filename must be either a binary executable, or a script starting with a line of the form:

```
#! interpreter [optional-arg]
```

`argv` is an array of argument strings passed to the new program. By convention, the first of these strings (i.e., `argv[0]`) should contain the filename associated with the file being executed. envp is an array of strings, conventionally of the form `key=value`, which are passed as environment to the new program. The `argv` and `envp` arrays must each include a null pointer at the end of the array.

The argument vector and environment can be accessed by the called program's main function, when it is defined as:

```
int main(int argc, char *argv[], char *envp[]);
```

By default, file descriptors remain open across an `execve()`. File descriptors that are marked close-on-exec are closed; see the description of `FD_CLOEXEC` in `fcntl(2)`.

**getuid, geteuid** (*get user identity*)

`getuid()` returns the real user ID of the calling process.

`geteuid()` returns the effective user ID of the calling process.

```
uid_t getuid(void);
uid_t geteuid(void);
```

**getgid, getegid** (*get group identity*)

`getgid()` returns the real group ID of the calling process.

`getegid()` returns the effective group ID of the calling process.

```
gid_t getgid(void);
gid_t getegid(void);
```

**setuid** (*set user identity*) `setuid()` sets the effective user ID of the calling process. If the calling process is privileged (more precisely: if the process has the `CAP_SETUID` capability in its user namespace), the real UID and saved set-user-ID are also set.

```
int setuid(uid_t uid);
```

Errors:

**EPERM** The user is not privileged (Linux: does not have the `CAP_SETUID` capability) and `uid` does not match the real UID or saved set-user-ID of the calling process.

**setreuid, setregid** (*set real and/or effective user or group ID*) `setreuid()` sets real and effective user IDs of the calling process.

Completely analogously, `setregid()` sets real and effective group ID's of the calling process, and all of the above holds with "group" instead of "user".

```
int setreuid(uid_t ruid, uid_t euid);
int setregid(gid_t rgid, gid_t egid);
```

**seteuid, setegid** (*set effective user or group ID*) `seteuid()` sets the effective user ID of the calling process. Unprivileged processes may only set the effective user ID to the real user ID, the effective user ID or the saved set-user-ID. Precisely the same holds for `setegid()` with "group" instead of "user".

```
int seteuid(uid_t euid);
int setegid(gid_t egid);
```

**getgroups** (*get list of supplementary group IDs*) `getgroups()` returns the supplementary group IDs of the calling process in list. The argument size should be set to the maximum number of items that can be stored in the buffer pointed to by list. If the calling process is a member of more than size supplementary groups, then an error results. It is unspecified whether the effective group ID of the calling process is included in the returned list. (Thus, an application should also call `getegid(2)` and add or remove the resulting value.)

```
int getgroups(int size, gid_t list[]);
```

If size is zero, list is not modified, but the total number of supplementary group IDs for the process is returned. This allows the caller to determine the size of a dynamically allocated list to be used in a further call to `getgroups()`.

On success, `getgroups()` returns the number of supplementary group IDs. On error, -1 is returned, and errno is set appropriately.

**setgroups** (*set list of supplementary group IDs*) `setgroups()` sets the supplementary group IDs for the calling process. Appropriate privileges are required (see the description of the EPERM error, below). The size argument specifies the number of supplementary group IDs in the buffer pointed to by list.

```
int setgroups(size_t size, const gid_t *list);
```

**getpid** (*get process identification*) `getpid()` returns the process ID (PID) of the calling process. (This is often used by routines that generate unique temporary filenames.)

```
pid_t getpid(void);
```

**getppid** (*get parent process identification*) `getppid()` returns the process ID of the parent of the calling process. This will be either the ID of the process that created this process using `fork()`, or, if that process has already terminated, the ID of the process to which this process has been reparented (either `init(1)` or a "subreaper" process defined via the `prctl(2)` `PR_SET_CHILD_SUBREAPER` operation).

```
pid_t getppid(void);
```

**times** (*get process times*) `times()` stores the current process times in the struct tms that buf points to. The struct tms is as defined in `<sys/times.h>`:

```
struct tms {
    clock_t tms_utime;  /* user time */
    clock_t tms_stime;  /* system time */
    clock_t tms_cutime; /* user time of children */
    clock_t tms_cstime; /* system time of children */
};

clock_t times(struct tms *buf);
```

The `tms_utime` field contains the CPU time spent executing instructions of the calling process. The `tms_stime` field contains the CPU time spent executing inside the kernel while performing tasks on behalf of the calling process.

The `tms_cutime` field contains the sum of the `tms_utime` and `tms_cutime` values for all waited-for terminated children. The `tms_cstime` field contains the sum of the `tms_stime` and `tms_cstime` values for all waited-for terminated children.

All times reported are in clock ticks.

**getrusage** (*get resource usage*)

```
int getrusage(int who, struct rusage *usage);
```

`getrusage()` returns resource usage measures for who, which can be one of the following:

`RUSAGE_SELF` Return resource usage statistics for the calling process, which is the sum of resources used by all threads in the process.

`RUSAGE_CHILDREN` Return resource usage statistics for all children of the calling process that have terminated and been waited for. These statistics will include the resources used by grandchildren, and further removed descendants, if all of the intervening descendants waited on their terminated children.

`RUSAGE_THREAD` Return resource usage statistics for the calling thread. The `_GNU_SOURCE` feature test macro must be defined (before including any header file) in order to obtain the definition of this constant from `<sys/resource.h>`.

The resource usages are returned in the structure pointed to by usage, which has the following form:

```
struct rusage {
    struct timeval ru_utime; /* user CPU time used */
    struct timeval ru_stime; /* system CPU time used */
    long   ru_maxrss;        /* maximum resident set size */
    long   ru_ixrss;         /* integral shared memory size */
    long   ru_idrss;         /* integral unshared data size */
    long   ru_isrss;         /* integral unshared stack size */
    long   ru_minflt;        /* page reclaims (soft page faults) */
    long   ru_majflt;        /* page faults (hard page faults) */
```

```
    long   ru_nswap;         /* swaps */
    long   ru_inblock;       /* block input operations */
    long   ru_oublock;       /* block output operations */
    long   ru_msgsnd;        /* IPC messages sent */
    long   ru_msgrcv;        /* IPC messages received */
    long   ru_nsignals;      /* signals received */
    long   ru_nvcsw;         /* voluntary context switches */
    long   ru_nivcsw;        /* involuntary context switches */
};
```

**getrlimit, setrlimit** (*get/set resource limits*) The `getrlimit()` and `setrlimit()` system calls get and set resource limits respectively. Each resource has an associated soft and hard limit, as defined by the rlimit structure:

```
struct rlimit {
    rlim_t rlim_cur;  /* Soft limit */
    rlim_t rlim_max;  /* Hard limit (ceiling for rlim_cur) */
};
```

The soft limit is the value that the kernel enforces for the corresponding resource. The hard limit acts as a ceiling for the soft limit: an unprivileged process may set only its soft limit to a value in the range from 0 up to the hard limit, and (irreversibly) lower its hard limit. A privileged process (under Linux: one with the `CAP_SYS_RESOURCE` capability in the initial user namespace) may make arbitrary changes to either limit value.

```
int getrlimit(int resource, struct rlimit *rlim);

int setrlimit(int resource, const struct rlimit *rlim);
```

The resource argument must be one of:

`RLIMIT_AS` This is the maximum size of the process's virtual memory (address space). The limit is specified in bytes, and is rounded down to the system page size. This limit affects calls to `brk(2)`, `mmap(2)`, and `mremap(2)`, which fail with the error `ENOMEM` upon exceeding this limit. In addition, automatic stack expansion fails (and generates a `SIGSEGV` that kills the process if no alternate stack has been made available via `sigaltstack(2)`).

`RLIMIT_CORE` This is the maximum size of a core file (see `core(5)`) in bytes that the process may dump. When 0 no core dump files are created. When nonzero, larger dumps are truncated to this size.

`RLIMIT_CPU` This is a limit, in seconds, on the amount of CPU time that the process can consume. When the process reaches the soft limit, it is sent a `SIGXCPU` signal. The default action for this signal is to terminate the process. However, the signal can be caught, and the handler can return control to the main program. If the process continues to consume CPU time, it will be sent `SIGXCPU` once per second until the hard limit is reached, at which time it is sent `SIGKILL`.

`RLIMIT_DATA` This is the maximum size of the process's data segment (initialized data, uninitialized data, and heap). The limit is specified in bytes, and is rounded down to the system page size. This limit affects calls to `brk(2)`, `sbrk(2)`, and (since Linux 4.7) `mmap(2)`, which fail with the error `ENOMEM` upon encountering the soft limit of this resource.

`RLIMIT_FSIZE` This is the maximum size in bytes of files that the process may create. Attempts to extend a file beyond this limit result in delivery of a `SIGXFSZ` signal. By default, this signal terminates a process, but a process can catch this signal instead, in which case the relevant system call (e.g., `write(2)`, `truncate(2)`) fails with the error `EFBIG`.

`RLIMIT_MEMLOCK` This is the maximum number of bytes of memory that may be locked into RAM. This limit is in effect rounded down to the nearest multiple of the system page size. This limit affects `mlock(2)`, `mlockall(2)`, and the `mmap(2)` `MAP_LOCKED` operation.

`RLIMIT_STACK` This is the maximum size of the process stack, in bytes. Upon reaching this limit, a `SIGSEGV` signal is generated. To handle this signal, a process must employ an alternate signal stack (`sigaltstack(2)`).

`RLIMIT_RSS` This is a limit (in bytes) on the process's resident set (the number of virtual pages resident in RAM). This limit has effect only in Linux 2.4.x, x $<$ 30, and there affects only calls to `madvise(2)` specifying `MADV_WILLNEED`.

`RLIMIT_NPROC` This is a limit on the number of extant process (or, more precisely on Linux, threads) for the real user ID of the calling process. So long as the current number of processes belonging to this process's real user ID is greater than or equal to this limit, `fork(2)` fails with the error `EAGAIN`.

`RLIMIT_NOFILE` This specifies a value one greater than the maximum file descriptor number that can be opened by this process. Attempts (`open(2)`, `pipe(2)`, `dup(2)`, etc.) to exceed this limit yield the error `EMFILE`.

**nice** (*change process priority*) `nice()` adds inc to the nice value for the calling thread. (A higher nice value means a low priority.)

```
int nice(int inc);
```

The range of the nice value is `+19` (low priority) to `-20` (high priority). Attempts to set a nice value outside the range are clamped to the range.

Traditionally, only a privileged process could lower the nice value (i.e., set a higher priority). However, since Linux 2.6.12, an unprivileged process can decrease the nice value of a target process that has a suitable `RLIMIT_NICE` soft limit; see `getrlimit(2)` for details.

**sched_yield** (*yield the processor*) sched_yield() causes the calling thread to relinquish the CPU. The thread is moved to the end of the queue for its static priority and a new thread gets to run.

```
int sched_yield(void);
```

If the calling thread is the only thread in the highest priority list at that time, it will continue to run after a call to `sched_yield()`. Avoid calling `sched_yield()` unnecessarily or inappropriately (e.g., when resources needed by other schedulable threads are still held by the caller), since doing so will result in unnecessary context switches, which will degrade system performance.

**getpriority, setpriority** (*get/set program scheduling priority*) The scheduling priority of the process, process group, or user, as indicated by which and who is obtained with the `getpriority()` call and set with the `setpriority()` call. The process attribute dealt with by these system calls is the same attribute (also known as the "nice" value) that is dealt with by `nice(2)`.

```
int getpriority(int which, id_t who);
int setpriority(int which, id_t who, int prio);
```

The value which is one of `PRIO_PROCESS`, `PRIO_PGRP`, or `PRIO_USER`, and who is interpreted relative to which (a process identifier for `PRIO_PROCESS`, process group identifier for `PRIO_PGRP`, and a user ID for `PRIO_USER`). A zero value for who denotes (respectively) the calling process, the process group of the calling process, or the real user ID of the calling process.

**setpgid, setpgrp** (*set process group*) `setpgid()` sets the PGID of the process specified by pid to pgid. If pid is zero, then the process ID of the calling process is used. If pgid is zero, then the PGID of the process specified by pid is made the same as its process ID. If `setpgid()` is used to move a process from one process group to another (as is done by some shells when creating pipelines), both process groups must be part of the same session (see `setsid(2)` and `credentials(7)`). In this case, the pgid specifies an existing process group to be joined and the session ID of that group must match the session ID of the joining process.

```
int setpgid(pid_t pid, pid_t pgid);
```

The System V-style `setpgrp()`, which takes no arguments, is equivalent to setpgid(0, 0).

**getpgid, getpgrp** (*get process group*) `getpgid()` returns the PGID of the process specified by pid. If pid is zero, the process ID of the calling process is used. (Retrieving the PGID of a process other than the caller is rarely necessary, and the POSIX.1 `getpgrp()` is preferred for that task.)

```
pid_t getpgid(pid_t pid);
```

The POSIX.1 version of `getpgrp()`, which takes no arguments, returns the PGID of the calling process.

**getsid** (*get session ID*) `getsid(0)` returns the session ID of the calling process. `getsid()` returns the session ID of the process with process ID pid. If pid is 0, `getsid()` returns the session ID of the calling process.

```
pid_t getsid(pid_t pid);
```

**setsid** (*creates a session and sets the process group ID*) `setsid()` creates a new session if the calling process is not a process group leader. The calling process is the leader of the new session (i.e., its session ID is made the same as its process ID). The calling process also becomes the process group leader of a new process group in the session (i.e., its process group ID is made the same as its process ID).

```
pid_t setsid(void);
```

**daemon** (*run in the background*) The `daemon()` function is for programs wishing to detach themselves from the controlling terminal and run in the background as system daemons.

```
int daemon(int nochdir, int noclose);
```

If nochdir is zero, `daemon()` changes the process's current working directory to the root directory (/); otherwise, the current working directory is left unchanged.

If noclose is zero, `daemon()` redirects standard input, standard output and standard error to /dev/null; otherwise, no changes are made to these file descriptors.

(This function forks, and if the `fork(2)` succeeds, the parent calls `_exit(2)`, so that further errors are seen by the child only.) On success `daemon()` returns zero. If an error occurs, `daemon()` returns -1 and sets `errno` to any of the errors specified for the `fork(2)` and `setsid(2)`.

## 6.3 Library calls (3)

**execl, execlp, execle, execv, execvp, execvpe** (*execute a file*) The `exec()` family of functions replaces the current process image with a new process image. The functions described in this manual page are front-ends for `execve(2)`.

```
int execl(const char *path, const char *arg, ...
/* (char  *) NULL */);
int execlp(const char *file, const char *arg, ...
/* (char  *) NULL */);
int execle(const char *path, const char *arg, ...
/*, (char *) NULL, char * const envp[] */);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],
char *const envp[]);
```

**system** (*execute a shell command*) The `system()` library function uses `fork(2)` to create a child process that executes the shell command specified in command using `execl(3)` as follows:

```
execl("/bin/sh", "sh", "-c", command, (char *) 0);
```

`system()` returns after the command has been completed.

**initgroups** (*initialize the supplementary group access list*) The `initgroups()` function initializes the group access list by reading the group database `/etc/group` and using all groups of which user is a member. The additional group group is also added to the list.

The user argument must be non-`NULL`.

```
int initgroups(const char *user, gid_t group);
```

## 6.4 File formats and conventions (5)

**magic** (*file command's magic pattern file*) This manual page documents the format of magic files as used by the `file(1)` command, version 5.34. The `file(1)` command identifies the type of a file using, among other tests, a test for whether the file contains certain "magic patterns". The database of these "magic patterns" is usually located in a binary file in `/usr/share/misc/magic.mgc` or a directory of source text magic pattern fragment files in `/usr/share/misc/magic`. The database specifies what patterns are to be tested for, what message or MIME type to print if a particular pattern is found, and additional information to extract from the file.

## 6.5 Miscellaneous (7)

**credentials** (*process identifiers*)

**Process ID (PID)** Each process has a unique nonnegative integer identifier that is assigned when the process is created using `fork(2)`. A process can obtain its PID using `getpid(2)`. A PID is represented using the type `pid_t` (defined in `<sys/types.h>`).

PIDs are used in a range of system calls to identify the process affected by the call, for example: `kill(2)`, `ptrace(2)`, `setpriority(2)` `setpgid(2)`, `setsid(2)`, `sigqueue(3)`, and `waitpid(2)`.

A process's PID is preserved across an `execve(2)`.

**Parent process ID (PPID)** A process's parent process ID identifies the process that created this process using `fork(2)`. A process can obtain its PPID using `getppid(2)`. A PPID is represented using the type `pid_t`.

A process's PPID is preserved across an `execve(2)`.

**Process group ID and session ID** Sessions and process groups are abstractions devised to support shell job control. A process group (sometimes called a "job") is a collection of processes that share the same process group ID; the shell creates a new process group for the process(es) used to execute single command or pipeline (e.g., the two processes created to execute the command `ls | wc` are placed in the same process group). A

process's group membership can be set using `setpgid(2)`. The process whose process ID is the same as its process group ID is the process group leader for that group.

A session is a collection of processes that share the same session ID. All of the members of a process group also have the same session ID (i.e., all of the members of a process group always belong to the same session, so that sessions and process groups form a strict two-level hierarchy of processes.) A new session is created when a process calls `setsid(2)`, which creates a new session whose session ID is the same as the PID of the process that called `setsid(2)`. The creator of the session is called the session leader.

**User and group identifiers** Each process has various associated user and group IDs. These IDs are integers, respectively represented using the types `uid_t` and `gid_t` (defined in `<sys/types.h>`).

On Linux, each process has the following user and group identifiers:

- Real user ID and real group ID. These IDs determine who owns the process. A process can obtain its real user (group) ID using `getuid(2)` (`getgid(2)`).
- Effective user ID and effective group ID. These IDs are used by the kernel to determine the permissions that the process will have when accessing shared resources such as message queues, shared memory, and semaphores. On most UNIX systems, these IDs also determine the permissions when accessing files. However, Linux uses the filesystem IDs described below for this task. A process can obtain its effective user (group) ID using `geteuid(2)` (`getegid(2)`).
- Saved set-user-ID and saved set-group-ID. These IDs are used in set-user-ID and set-group-ID programs to save a copy of the corresponding effective IDs that were set when the program was executed (see `execve(2)`). A set-user-ID program can assume and drop privileges by switching its effective user ID back and forth between the values in its real user ID and saved set-user-ID. This switching is done via calls to `seteuid(2)`, `setreuid(2)`, or `setresuid(2)`. A set-group-ID program performs the analogous tasks using `setegid(2)`, `setregid(2)`, or `setresgid(2)`. A process can obtain its saved set-user-ID (set-group-ID) using `getresuid(2)` (`getresgid(2)`)
- Filesystem user ID and filesystem group ID (Linux-specific). These IDs, in conjunction with the supplementary group IDs described below, are used to determine permissions for accessing files; see `path_resolution(7)` for details. Whenever a process's effective user (group) ID is changed, the kernel also automatically changes the filesystem user (group) ID to the same value. Consequently, the filesystem IDs normally have the same values as the corresponding effective ID, and the semantics for file-permission checks are thus the same on Linux as on other UNIX systems. The filesystem IDs can be made to differ from the effective IDs by calling `setfsuid(2)` and `setfsgid(2)`.
- Supplementary group IDs. This is a set of additional group IDs that are used for permission checks when accessing files and other shared resources. On Linux kernels before 2.6.4, a process can be a member of up to 32 supplementary groups; since kernel 2.6.4, a process can be a member of up to 65536 supplementary groups. The call sysconf(`_SC_NGROUPS_MAX`) can be used to determine the number of supplementary groups of which a process may be a member. A process can obtain its set of supplementary group IDs using `getgroups(2)`, and can modify the set using `setgroups(2)`.

A child process created by `fork(2)` inherits copies of its parent's user and groups IDs. During an `execve(2)`, a process's real user and group ID and supplementary group IDs are preserved; the effective and saved set IDs may be changed, as described in `execve(2)`.

**capabilities** (*overview of Linux capabilities*) For the purpose of performing permission checks, traditional UNIX implementations distinguish two categories of processes: privileged processes (whose effective user ID is 0, referred to as superuser or root), and unprivileged processes (whose effective UID is nonzero). Privileged processes bypass all kernel permission checks,

while unprivileged processes are subject to full permission checking based on the process's credentials (usually: effective UID, effective GID, and supplementary group list).

Starting with kernel 2.2, Linux divides the privileges traditionally associated with superuser into distinct units, known as capabilities, which can be independently enabled and disabled. Capabilities are a per-thread attribute.

# 7 I/O operations

## 7.1 System calls (2)

**open** (*open and possibly create a file*) The `open()` system call opens the file specified by pathname. If the specified file does not exist, it may optionally (if `O_CREAT` is specified in flags) be created by `open()`.

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

The return value of `open()` is a file descriptor, a small, nonnegative integer that is used in subsequent system calls (`read(2)`, `write(2)`, `lseek(2)`, `fcntl(2)`, etc.) to refer to the open file. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

By default, the new file descriptor is set to remain open across an `execve(2)` (i.e., the `FD_CLOEXEC` file descriptor flag described in `fcntl(2)` is initially disabled); the `O_CLOEXEC` flag, described below, can be used to change this default. The file offset is set to the beginning of the file (see `lseek(2)`).

A call to `open()` creates a new open file description, an entry in the system-wide table of open files. The open file description records the file offset and the file status flags (see below). A file descriptor is a reference to an open file description; this reference is unaffected if pathname is subsequently removed or modified to refer to a different file. For further details on open file descriptions, see NOTES.

The argument flags must include one of the following access modes: `O_RDONLY`, `O_WRONLY`, or `O_RDWR`. These request opening the file read-only, write-only, or read/write, respectively.

In addition, zero or more file creation flags and file status flags can be bitwise-or'd in flags. The full list of file creation flags and file status flags is as follows:

`O_CREAT` If pathname does not exist, create it as a regular file. The owner (user ID) of the new file is set to the effective user ID of the process.

`O_EXCL` Ensure that this call creates the file: if this flag is specified in conjunction with `O_CREAT`, and pathname already exists, then `open()` fails with the error `EEXIST`.

`O_TRUNC` If the file already exists and is a regular file and the access mode allows writing (i.e., is `O_RDWR` or `O_WRONLY`) it will be truncated to length 0.

`O_APPEND` The file is opened in append mode. Before each `write(2)`, the file offset is positioned at the end of the file, as if with `lseek(2)`. The modification of the file offset and the write operation are performed as a single atomic step.

`O_NONBLOCK` **or** `O_NDELAY` When possible, the file is opened in nonblocking mode. Neither the `open()` nor any subsequent operations on the file descriptor which is returned will cause the calling process to wait.

`O_SYNC` Write operations on the file will complete according to the requirements of synchronized I/O file integrity completion (by contrast with the synchronized I/O data integrity completion provided by `O_DSYNC`.)

The term **open file description** is the one used by POSIX to refer to the entries in the system- wide table of open files. In other contexts, this object is variously also called an "open

file object", a "file handle", an "open file table entry", or—in kernel-developer parlance—a struct file.

When a file descriptor is duplicated (using `dup(2)` or similar), the duplicate refers to the same open file description as the original file descriptor, and the two file descriptors consequently share the file offset and file status flags. Such sharing can also occur between processes: a child process created via `fork(2)` inherits duplicates of its parent's file descriptors, and those duplicates refer to the same open file descriptions.

Each `open()` of a file creates a new open file description; thus, there may be multiple open file descriptions corresponding to a file inode.

**creat** (*open and create a file*) A call to `creat()` is equivalent to calling `open()` with flags equal to `O_CREAT|O_WRONLY|O_TRUNC`.

```
int creat(const char *pathname, mode_t mode);
```

**close** (*close a file descriptor*) `close()` closes a file descriptor, so that it no longer refers to any file and may be reused. Any record locks (see `fcntl(2)`) held on the file it was associated with, and owned by the process, are removed (regardless of the file descriptor that was used to obtain the lock).

```
int close(int fd);
```

If fd is the last file descriptor referring to the underlying open file description (see `open(2)`), the resources associated with the open file description are freed; if the file descriptor was the last reference to a file which has been removed using `unlink(2)`, the file is deleted.

**read** (*read from a file descriptor*) `read()` attempts to read up to `count` bytes from file descriptor `fd` into the buffer starting at `buf`.

```
ssize_t read(int fd, void *buf, size_t count);
```

On files that support seeking, the read operation commences at the file offset, and the file offset is incremented by the number of bytes read. If the file offset is at or past the end of file, no bytes are read, and `read()` returns zero.

On error, -1 is returned, and `errno` is set appropriately. In this case, it is left unspecified whether the file position (if any) changes.

**write** (*write to a file descriptor*) `write()` writes up to `count` bytes from the buffer starting at `buf` to the file referred to by the file descriptor `fd`.

```
ssize_t write(int fd, const void *buf, size_t count);
```

The number of bytes written may be less than count if, for example, there is insufficient space on the underlying physical medium, or the `RLIMIT_FSIZE` resource limit is encountered (see `setrlimit(2)`), or the call was interrupted by a signal handler after having written less than count bytes. (See also `pipe(7)`.)

For a seekable file (i.e., one to which `lseek(2)` may be applied, for example, a regular file) writing takes place at the file offset, and the file offset is incremented by the number of bytes actually written. If the file was `open(2)`-ed with `O_APPEND`, the file offset is first set to the end of the file before writing. The adjustment of the file offset and the write operation are performed as an atomic step.

**lseek** (*reposition read/write file offset*)

```
off_t lseek(int fd, off_t offset, int whence);
```

`lseek()` repositions the file offset of the open file description associated with the file descriptor `fd` to the argument `offset` according to the directive `whence` as follows:

`SEEK_SET` The file offset is set to offset bytes.

`SEEK_CUR` The file offset is set to its current location plus offset bytes.

`SEEK_END` The file offset is set to the size of the file plus offset bytes.

`lseek()` allows the file offset to be set beyond the end of the file (but this does not change the size of the file). If data is later written at this point, subsequent reads of the data in the gap (a "hole") return null bytes (`\0`) until data is actually written into the gap.

**dup** (*duplicate a file descriptor*) The `dup()` system call creates a copy of the file descriptor `oldfd`, using the lowest-numbered unused file descriptor for the new descriptor.

```
int dup(int oldfd);
```

After a successful return, the old and new file descriptors may be used interchangeably. They refer to the same open file description (see `open(2)`) and thus share file offset and file status flags; for example, if the file offset is modified by using `lseek(2)` on one of the file descriptors, the offset is also changed for the other.

**dup2** (*duplicate a file descriptor*) The `dup2()` system call performs the same task as `dup()`, but instead of using the lowest-numbered unused file descriptor, it uses the file descriptor number specified in `newfd`. If the file descriptor `newfd` was previously open, it is silently closed before being reused.

```
int dup2(int oldfd, int newfd);
```

**fcntl** (*manipulate file descriptor*) `fcntl()` performs one of the operations described below on the open file descriptor `fd`. The operation is determined by `cmd`.

```
int fcntl(int fd, int cmd, ... /* arg */ );
```

`fcntl()` can take an optional third argument. Whether or not this argument is required is determined by `cmd`. The required argument type is indicated in parentheses after each `cmd` name (in most cases, the required type is int, and we identify the argument using the name arg), or void is specified if the argument is not required.

`F_DUPFD (int)` Duplicate the file descriptor `fd` using the lowest-numbered available file descriptor greater than or equal to `arg`. This is different from `dup2(2)`, which uses exactly the file descriptor specified. On success, the new file descriptor is returned.

`F_GETFD (void)` Return (as the function result) the file descriptor flags; `arg` is ignored.

`F_SETFD (int)` Set the file descriptor flags to the value specified by `arg`.

`F_GETFL (void)` Return (as the function result) the file access mode and the file status flags; `arg` is ignored.

`F_SETFL (int)` Set the file status flags to the value specified by arg. On Linux, this command can change only the `O_APPEND`, `O_ASYNC`, `O_DIRECT`, `O_NOATIME`, and `O_NONBLOCK` flags.

`F_SETLK`, `F_SETLKW`, and `F_GETLK` are used to acquire, release, and test for the existence of record locks (also known as byte-range, file-segment, or file-region locks). The third argument, lock, is a pointer to a structure that has at least the following fields (in unspecified order).

```
struct flock {
    ...
    short l_type;    /* Type of lock: F_RDLCK,
F_WRLCK, F_UNLCK */
    short l_whence;  /* How to interpret l_start:
```

```
SEEK_SET, SEEK_CUR, SEEK_END */
    off_t l_start;   /* Starting offset for lock */
    off_t l_len;     /* Number of bytes to lock */
    pid_t l_pid;     /* PID of process blocking our lock
(set by F_GETLK and F_OFD_GETLK) */
    ...
};
```

The `l_whence`, `l_start`, and `l_len` fields of this structure specify the range of bytes we wish to lock. Bytes past the end of the file may be locked, but not bytes before the start of the file.

`l_start` is the starting offset for the lock, and is interpreted relative to either: the start of the file (if `l_whence` is `SEEK_SET`); the current file offset (if `l_whence` is `SEEK_CUR`); or the end of the file (if `l_whence` is `SEEK_END`).

`l_len` specifies the number of bytes to be locked. If `l_len` is positive, then the range to be locked covers bytes `l_start` up to and including `l_start+l_len-1`. Specifying 0 for `l_len` has the special meaning: lock all bytes starting at the location specified by `l_whence` and `l_start` through to the end of file, no matter how large the file grows.

The `l_type` field can be used to place a read (`F_RDLCK`) or a write (`F_WRLCK`) lock on a file. Any number of processes may hold a read lock (shared lock) on a file region, but only one process may hold a write lock (exclusive lock). An exclusive lock excludes all other locks, both shared and exclusive.

`F_SETLK (struct flock *)` Acquire a lock (when `l_type` is `F_RDLCK` or `F_WRLCK`) or release a lock (when `l_type` is `F_UNLCK`) on the bytes specified by the `l_whence`, `l_start`, and `l_len` fields of lock. If a conflicting lock is held by another process, this call returns -1 and sets `errno` to `EACCES` or `EAGAIN`.

`F_SETLKW (struct flock *)` As for `F_SETLK`, but if a conflicting lock is held on the file, then wait for that lock to be released. If a signal is caught while waiting, then the call is interrupted and (after the signal handler has returned) returns immediately (with return value -1 and errno set to `EINTR`; see `signal(7)`).

`F_GETLK (struct flock *)` On input to this call, lock describes a lock we would like to place on the file. If the lock could be placed, `fcntl()` does not actually place it, but returns `F_UNLCK` in the `l_type` field of lock and leaves the other fields of the structure unchanged.

**ioctl** (*control device*) The `ioctl()` system call manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g., terminals) may be controlled with `ioctl()` requests. The argument `fd` must be an open file descriptor.

```
int ioctl(int fd, unsigned long request, ...);
```

The second argument is a device-dependent request code. The third argument is an untyped pointer to memory.

# 8 Operations on files

## 8.1 User commands (1)

**ls** (*list directory contents*) List information about the FILEs (the current directory by default).

**mktemp** (*create a temporary file or directory*) Create a temporary file or directory, safely, and print its name. TEMPLATE must contain at least 3 consecutive 'X's in last component. If `TEMPLATE` is not specified, use `tmp.XXXXXXXXXX`, and `--tmpdir` is implied. Files are created `u+rw`, and directories `u+rwx`, minus `umask` restrictions.

**getfacl** (*get file access control lists*) For each file, getfacl displays the file name, owner, the group, and the Access Control List (ACL). If a directory has a default ACL, getfacl also displays the default ACL.

**setfacl** (*set file access control lists*) This utility sets Access Control Lists (ACLs) of files and directories. On the command line, a sequence of commands is followed by a sequence of files (which in turn can be followed by another sequence of commands, ...).

**chacl** (*change the access control list of a file or directory*) `chacl` is an IRIX-compatibility command, and is maintained for those users who are familiar with its use from either XFS or IRIX.

`chacl` changes the ACL(s) for a file or directory. The ACL(s) specified are applied to each file in the `pathname` arguments.

## 8.2 System calls (2)

**stat, fstat, lstat** (*get file status*) These functions return information about a file, in the buffer pointed to by `statbuf`. No permissions are required on the file itself, but—in the case of `stat()`, `fstatat()`, and `lstat()`-execute (search) permission is required on all of the directories in `pathname` that lead to the file.

```
int stat(const char *pathname, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);
int lstat(const char *pathname, struct stat *statbuf);
```

`stat()` and `fstatat()` retrieve information about the file pointed to by `pathname`; the differences for `fstatat()` are described below.

`lstat()` is identical to `stat()`, except that if `pathname` is a symbolic link, then it returns information about the link itself, not the file that it refers to.

`fstat()` is identical to `stat()`, except that the file about which information is to be retrieved is specified by the file descriptor `fd`.

**The stat structure** All of these system calls return a stat structure, which contains the following fields:

```
struct stat {
    dev_t     st_dev;         /* ID of device containing file */
    ino_t     st_ino;         /* Inode number */
    mode_t    st_mode;        /* File type and mode */
    nlink_t   st_nlink;       /* Number of hard links */
    uid_t     st_uid;         /* User ID of owner */
    gid_t     st_gid;         /* Group ID of owner */
    dev_t     st_rdev;        /* Device ID (if special file) */
    off_t     st_size;        /* Total size, in bytes */
    blksize_t st_blksize;     /* Block size for filesystem I/O */
    blkcnt_t  st_blocks;      /* Number of 512B blocks allocated */

    /* Since Linux 2.6, the kernel supports nanosecond
       precision for the following timestamp fields.
       For the details before Linux 2.6, see NOTES. */

    struct timespec st_atim;  /* Time of last access */
    struct timespec st_mtim;  /* Time of last modification */
    struct timespec st_ctim;  /* Time of last status change */
};
```

**access** (*check user's permissions for a file*) `access()` checks whether the calling process can access the file `pathname`. If `pathname` is a symbolic link, it is dereferenced.

```
int access(const char *pathname, int mode);
```

The mode specifies the accessibility check(s) to be performed, and is either the value `F_OK`, or a mask consisting of the bitwise OR of one or more of `R_OK`, `W_OK`, and `X_OK`. `F_OK` tests for the existence of the file. `R_OK`, `W_OK`, and `X_OK` test whether the file exists and grants read, write, and execute permissions, respectively.

The check is done using the calling process's real UID and GID, rather than the effective IDs as is done when actually attempting an operation (e.g., `open(2)`) on the file. In other words, access() does not answer the "can I read/write/execute this file?" question. It answers a slightly different question: "(assuming I'm a setuid binary) can the user who invoked me read/write/execute this file?", which gives set-user-ID programs the possibility to prevent malicious users from causing them to read files which users shouldn't be able to read.

On success (all requested permissions granted, or mode is `F_OK` and the file exists), zero is returned.

**umask** (*set file mode creation mask*) `umask()` sets the calling process's file mode creation mask (umask) to `mask & 0777` (i.e., only the file permission bits of mask are used), and returns the previous value of the mask.

```
mode_t umask(mode_t mask);
```

The umask is used by `open(2)`, `mkdir(2)`, and other system calls that create files to modify the permissions placed on newly created files or directories. Specifically, permissions in the umask are turned off from the mode argument to `open(2)` and `mkdir(2)`.

Alternatively, if the parent directory has a default ACL (see `acl(5)`), the umask is ignored, the default ACL is inherited, the permission bits are set based on the inherited ACL, and permission bits absent in the mode argument are turned off. For example, the following default ACL is equivalent to a `umask` of 022:

```
u::rwx,g::r-x,o::r-x
```

Combining the effect of this default ACL with a mode argument of 0666 (`rw-rw-rw-`), the resulting file permissions would be 0644 (`rw-r--r--`).

**chown, fchown, lchown** (*change ownership of a file*) These system calls change the owner and group of a file. The `chown()`, `fchown()`, and `lchown()` system calls differ only in how the file is specified:

- `chown()` changes the ownership of the file specified by `pathname`, which is dereferenced if it is a symbolic link.
- `fchown()` changes the ownership of the file referred to by the open file descriptor `fd`.
- `lchown()` is like `chown()`, but does not dereference symbolic links.

```
int chown(const char *pathname, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int lchown(const char *pathname, uid_t owner, gid_t group);
```

Only a privileged process (Linux: one with the `CAP_CHOWN` capability) may change the owner of a file. The owner of a file may change the group of the file to any group of which that owner is a member. A privileged process (Linux: with `CAP_CHOWN`) may change the group arbitrarily.

If the owner or group is specified as -1, then that ID is not changed.

**chmod, fchmod** (*change permissions of a file*) The `chmod()` and `fchmod()` system calls change a files mode bits. (The file mode consists of the file permission bits plus the set-user-ID, set-group-ID, and sticky bits.) These system calls differ only in how the file is specified:

- `chmod()` changes the mode of the file specified whose pathname is given in `pathname`, which is dereferenced if it is a symbolic link.
- `fchmod()` changes the mode of the file referred to by the open file descriptor `fd`.

```
int chmod(const char *pathname, mode_t mode);
int fchmod(int fd, mode_t mode);
```

**truncate, ftruncate** (*truncate a file to a specified length*) The `truncate()` and `ftruncate()` functions cause the regular file named by `path` or referenced by `fd` to be truncated to a size of precisely `length` bytes.

```
int truncate(const char *path, off_t length);
int ftruncate(int fd, off_t length);
```

If the file previously was larger than this size, the extra data is lost. If the file previously was shorter, it is extended, and the extended part reads as null bytes (`\0`).

The file offset is not changed.

If the size changed, then the `st_ctime` and `st_mtime` fields (respectively, time of last status change and time of last modification; see `inode(7)`) for the file are updated, and the set-user-ID and set-group-ID mode bits may be cleared.

**link** (*make a new name for a file*) `link()` creates a new link (also known as a hard link) to an existing file.

```
int link(const char *oldpath, const char *newpath);
```

If `newpath` exists, it will not be overwritten.

This new name may be used exactly as the old one for any operation; both names refer to the same file (and so have the same permissions and ownership) and it is impossible to tell which name was the "original".

**unlink** (*delete a name and possibly the file it refers to*) `unlink()` deletes a name from the filesystem. If that name was the last link to a file and no processes have the file open, the file is deleted and the space it was using is made available for reuse.

```
int unlink(const char *pathname);
```

If the name was the last link to a file but any processes still have the file open, the file will remain in existence until the last file descriptor referring to it is closed.

If the name referred to a symbolic link, the link is removed.

If the name referred to a socket, FIFO, or device, the name for it is removed but processes which have the object open may continue to use it.

**rename** (*change the name or location of a file*) `rename()` renames a file, moving it between directories if required. Any other hard links to the file (as created using `link(2)`) are unaffected. Open file descriptors for `oldpath` are also unaffected.

```
int rename(const char *oldpath, const char *newpath);
```

If `newpath` already exists, it will be atomically replaced, so that there is no point at which another process attempting to access `newpath` will find it missing.

**utime** (*change file last access and modification times*)

```
int utime(const char *filename, const struct utimbuf *times);
```

The `utime()` system call changes the access and modification times of the inode specified by `filename` to the `actime` and `modtime` fields of `times` respectively.

If times is `NULL`, then the access and modification times of the file are set to the current time.

Changing timestamps is permitted when: either the process has appropriate privileges, or the effective user ID equals the user ID of the file, or `times` is `NULL` and the process has write permission for the file.

The `utimbuf` structure is:

```
struct utimbuf {
    time_t actime;       /* access time */
    time_t modtime;      /* modification time */
};
```

The `utime()` system call allows specification of timestamps with a resolution of 1 second.

**utimes** (*change file last access and modification times*)

```
int utimes(const char *filename, const struct timeval times[2]);
```

The `utimes()` system call is similar, but the `times` argument refers to an array rather than a structure. The elements of this array are `timeval` structures, which allow a precision of 1 microsecond for specifying timestamps. The `timeval` structure is:

```
struct timeval {
    long tv_sec;        /* seconds */
    long tv_usec;       /* microseconds */
};
```

`times[0]` specifies the new access time, and `times[1]` specifies the new modification time. If `times` is `NULL`, then analogously to `utime()`, the access and modification times of the file are set to the current time.

**symlink** (*make a new name for a file*) `symlink()` creates a symbolic link named `linkpath` which contains the string `target`.

```
int symlink(const char *target, const char *linkpath);
```

Symbolic links are interpreted at run time as if the contents of the link had been substituted into the path being followed to find a file or directory.

Symbolic links may contain `..` path components, which (if used at the start of the link) refer to the parent directories of that in which the link resides.

A symbolic link (also known as a soft link) may point to an existing file or to a nonexistent one; the latter case is known as a dangling link.

The permissions of a symbolic link are irrelevant; the ownership is ignored when following the link, but is checked when removal or renaming of the link is requested and the link is in a directory with the sticky bit (`S_ISVTX`) set.

If `linkpath` exists, it will not be overwritten.

**readlink** read value of a symbolic link `readlink()` places the contents of the symbolic link `pathname` in the buffer `buf`, which has size `bufsiz`. `readlink()` does not append a null byte to `buf`. It will (silently) truncate the contents (to a length of `bufsiz` characters), in case the buffer is too small to hold all of the contents.

```
ssize_t readlink(const char *pathname, char *buf, size_t bufsiz);
```

**mkdir** (*create a directory*) `mkdir()` attempts to create a directory named `pathname`.

```
int mkdir(const char *pathname, mode_t mode);
```

The argument `mode` specifies the mode for the new directory (see `inode(7)`). It is modified by the process's `umask` in the usual way: in the absence of a default ACL, the mode of the created directory is (`mode & ~umask & 0777`).

**rmdir** (*delete a directory*) `rmdir()` deletes a directory, which must be empty.

```
int rmdir(const char *pathname);
```

**chdir** (*change working directory*) `chdir()` changes the current working directory of the calling process to the directory specified in `path`.

```
int chdir(const char *path);
```

**fchdir** (*change working directory*) `fchdir()` is identical to `chdir()`; the only difference is that the directory is given as an open file descriptor.

```
int fchdir(int fd);
```

**chroot** (*change root directory*) `chroot()` changes the root directory of the calling process to that specified in `path`. This directory will be used for pathnames beginning with `/`. The root directory is inherited by all children of the calling process.

Only a privileged process (Linux: one with the `CAP_SYS_CHROOT` capability in its user namespace) may call `chroot()`.

**sync** (*commit filesystem caches to disk*) `sync()` causes all pending modifications to filesystem metadata and cached file data to be written to the underlying filesystems.

```
void sync(void);
```

**fsync** (*synchronize a file's in-core state with storage device*) `fsync()` transfers ("flushes") all modified in-core data of (i.e., modified buffer cache pages for) the file referred to by the file descriptor `fd` to the disk device (or other permanent storage device) so that all changed information can be retrieved even if the system crashes or is rebooted. This includes writing through or flushing a disk cache if present. The call blocks until the device reports that the transfer has completed.

```
int fsync(int fd);
```

As well as flushing the file data, `fsync()` also flushes the metadata information associated with the file (see `inode(7)`).

Calling `fsync()` does not necessarily ensure that the entry in the directory containing the file has also reached disk. For that an explicit `fsync()` on a file descriptor for the directory is also needed.

**fdatasync** (*synchronize a file's in-core state with storage device*) `fdatasync()` is similar to `fsync()`, but does not flush modified metadata unless that metadata is needed in order to allow a subsequent data retrieval to be correctly handled.

```
int fdatasync(int fd);
```

**mknod** (*create a special or ordinary file*) The system call `mknod()` creates a filesystem node (file, device special file, or named pipe) named `pathname`, with attributes specified by `mode` and `dev`.

```
int mknod(const char *pathname, mode_t mode, dev_t dev);
```

The `mode` argument specifies both the file mode to use and the type of node to be created. It should be a combination (using bitwise OR) of one of the file types listed below and zero or more of the file mode bits listed in `inode(7)`.

The file type must be one of `S_IFREG`, `S_IFCHR`, `S_IFBLK`, `S_IFIFO`, or `S_IFSOCK` to specify a regular file (which will be created empty), character special file, block special file, FIFO (named pipe), or UNIX domain socket, respectively. (Zero file type is equivalent to type `S_IFREG`.)

## 8.3 Library calls (3)

**remove** (*remove a file or directory*) `remove()` deletes a name from the filesystem. It calls `unlink(2)` for files, and `rmdir(2)` for directories.

```
int remove(const char *pathname);
```

**mkstemp** (*create a unique temporary file*) The `mkstemp()` function generates a unique temporary filename from `template`, creates and opens the file, and returns an open file descriptor for the file.

```
int mkstemp(char *template);
```

The last six characters of `template` must be `XXXXXX` and these are replaced with a string that makes the filename unique. Since it will be modified, `template` must not be a string constant, but should be declared as a character array.

The file is created with permissions 0600, that is, read plus write for owner only. The returned file descriptor provides both read and write access to the file. The file is opened with the `open(2)` `O_EXCL` flag, guaranteeing that the caller is the process that creates the file.

**mktemp** (*make a unique temporary filename*) Never use this function; see `BUGS`.

**tmpnam, tmpnam_r** (*create a name for a temporary file*) Note: avoid using these functions; use `mkstemp(3)` or `tmpfile(3)` instead.

**tempnam** (*create a name for a temporary file*) Never use this function. Use `mkstemp(3)` or `tmpfile(3)` instead.

**opendir** (*open a directory*) The `opendir()` function opens a directory stream corresponding to the directory `name`, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory.

```
DIR *opendir(const char *name);
```

**closedir** (*close a directory*) The `closedir()` function closes the directory stream associated with `dirp`. A successful call to `closedir()` also closes the underlying file descriptor associated with `dirp`. The directory stream descriptor `dirp` is not available after this call.

```
int closedir(DIR *dirp);
```

**readdir** (*read a directory*) The `readdir()` function returns a pointer to a `dirent` structure representing the next directory entry in the directory stream pointed to by `dirp`. It returns `NULL` on reaching the end of the directory stream or if an error occurred.

In the glibc implementation, the `dirent` structure is defined as follows:

```
struct dirent {
    ino_t          d_ino;       /* Inode number */
    off_t          d_off;       /* Not an offset; see below */
    unsigned short d_reclen;    /* Length of this record */
```

```
    unsigned char  d_type;      /* Type of file; not supported
   by all filesystem types */
    char           d_name[256]; /* Null-terminated filename */
};
```

**getcwd, getwd** (*get current working directory*) These functions return a null-terminated string containing an absolute pathname that is the current working directory of the calling process. The pathname is returned as the function result and via the argument `buf`, if present.

```
char *getcwd(char *buf, size_t size);
char *getwd(char *buf);
```

**mkfifo** (*make a FIFO special file (a named pipe)*) `mkfifo()` makes a FIFO special file with name `pathname`. `mode` specifies the FIFO's permissions.

```
int mkfifo(const char *pathname, mode_t mode);
```

## 8.4 File formats and conventions (5)

**acl** (*Access Control Lists*) This manual page describes POSIX Access Control Lists, which are used to define more fine- grained discretionary access rights for files and directories.

Every object can be thought of as having associated with it an ACL that governs the discretionary access to that object; this ACL is referred to as an access ACL. In addition, a directory may have an associated ACL that governs the initial access ACL for objects created within that directory; this ACL is referred to as a default ACL.

An ACL consists of a set of ACL entries. An ACL entry specifies the access permissions on the associated object for an individual user or a group of users as a combination of read, write and search/execute permissions.

The long text form contains one ACL entry per line. In addition, a number sign (`#`) may start a comment that extends until the end of the line. If an `ACL_USER`, `ACL_GROUP_OBJ` or `ACL_GROUP` ACL entry contains permissions that are not also contained in the `ACL_MASK` entry, the entry is followed by a number sign, the string "effective:", and the effective access permissions defined by that entry. This is an example of the long text form:

```
user::rw-
user:lisa:rw-         #effective:r--
group::r--
group:toolies:rw-     #effective:r--
mask::r--
other::r--
```

These are examples of the short text form:

```
u::rw-,u:lisa:rw-,g::r--,g:toolies:rw-,m::r--,o::r--
```

```
g:toolies:rw,u:lisa:rw,u::wr,g::r,o::r,m::r
```

## 8.5 Miscellaneous (7)

**inode** (*file inode information*) Each file has an inode containing metadata about the file. An application can retrieve this metadata using `stat(2)` (or related calls), which returns a `stat` structure, or `statx(2)`, which returns a `statx` structure.

...additional macros are defined by POSIX to allow the test of the file type in `st_mode` to be written more concisely:

```
S_ISREG(m)  is it a regular file?
S_ISDIR(m)  directory?
S_ISCHR(m)  character device?
S_ISBLK(m)  block device?
S_ISFIFO(m) FIFO (named pipe)?
S_ISLNK(m)  symbolic link?  (Not in POSIX.1-1996.)
S_ISSOCK(m) socket?  (Not in POSIX.1-1996.)
```

The following mask values are defined for the file mode component of the `st_mode` field:

```
S_ISUID     04000   set-user-ID bit
S_ISGID     02000   set-group-ID bit (see below)
S_ISVTX     01000   sticky bit (see below)

S_IRWXU     00700   owner has read, write, and execute permission
S_IRUSR     00400   owner has read permission
S_IWUSR     00200   owner has write permission
S_IXUSR     00100   owner has execute permission

S_IRWXG     00070   group has read, write, and execute permission
S_IRGRP     00040   group has read permission
S_IWGRP     00020   group has write permission
S_IXGRP     00010   group has execute permission

S_IRWXO     00007   others (not in group) have read,  write,  and
    execute permission
S_IROTH     00004   others have read permission
S_IWOTH     00002   others have write permission
S_IXOTH     00001   others have execute permission
```

# 9 Filesystems

## 9.1 File formats and conventions (5)

**filesystems** (*Linux filesystem types: ext, ext2, nfs, ntfs ...*) When, as is customary, the proc filesystem is mounted on `/proc`, you can find in the file `/proc/filesystems` which filesystems your kernel currently supports; see `proc(5)` for more details. If you need a currently unsupported filesystem, insert the corresponding module or recompile the kernel.

In order to use a filesystem, you have to mount it; see `mount(8)`.

Below a short description of the available or historically available filesystems in the Linux kernel.

**ext** is an elaborate extension of the minix filesystem. It has been completely superseded by the second version of the extended filesystem (ext2) and has been removed from the kernel (in 2.1.21).

**ext2** is the high performance disk filesystem used by Linux for fixed disks as well as removable media. The second extended filesystem was designed as an extension of the extended filesystem (ext). See ext2(5).

**ext3** is a journaling version of the ext2 filesystem. It is easy to switch back and forth between ext2 and ext3. See ext3(5).

**ext4** is a set of upgrades to ext3 including substantial performance and reliability enhancements, plus large increases in volume, file, and directory size limits. See ext4(5).

**ntfs** replaces Microsoft Window's FAT filesystems (VFAT, FAT32). It has reliability, performance, and space-utilization enhancements plus features like ACLs, journaling, encryption, and so on.

**Reiserfs** is a journaling filesystem, designed by Hans Reiser, that was integrated into Linux in kernel 2.4.1.

**XFS** is a journaling filesystem, developed by SGI, that was integrated into Linux in kernel 2.4.20.

### 9.2 System management commands (8)

**fsck** (*check and repair a Linux filesystem*) `fsck` is used to check and optionally repair one or more Linux filesystems. `filesys` can be a device name (e.g. `/dev/hdc1`, `/dev/sdb2`), a mount point (e.g. `/`, `/usr`, `/home`), or an ext2 label or UUID specifier.

**lvm** (*LVM2 tools*) The Logical Volume Manager (LVM) provides tools to create virtual block devices from physical devices. Virtual devices may be easier to manage than physical devices, and can have capabilities beyond what the physical devices provide themselves. A Volume Group (VG) is a collection of one or more physical devices, each called a Physical Volume (PV). A Logical Volume (LV) is a virtual block device that can be used by the system or applications. Each block of data in an LV is stored on one or more PV in the VG, according to algorithms implemented by Device Mapper (DM) in the kernel.

The lvm command, and other commands listed below, are the command-line tools for LVM. A separate manual page describes each command in detail.

## 10 Inter process communication

### 10.1 System calls (2)

**pipe** (*create pipe*) `pipe()` creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array `pipefd` is used to return two file descriptors referring to the ends of the pipe. `pipefd[0]` refers to the read end of the pipe. `pipefd[1]` refers to the write end of the pipe. Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe. For further details, see `pipe(7)`.

```
int pipe(int pipefd[2]);
```

The following program creates a pipe, and then `fork(2)`-s to create a child process; the child inherits a duplicate set of file descriptors that refer to the same pipe. After the `fork(2)`, each process closes the file descriptors that it doesn't need for the pipe (see `pipe(7)`). The parent then writes the string contained in the program's command-line argument to the pipe, and the child reads this string a byte at a time from the pipe and echoes it on standard output.

```
int pipefd[2];
pid_t cpid;
char buf;

if (argc != 2) {
    fprintf(stderr, "Usage: %s <string>\n", argv[0]);
    exit(EXIT_FAILURE);
}

if (pipe(pipefd) == -1) {
    perror("pipe");
    exit(EXIT_FAILURE);
}

cpid = fork();
if (cpid == -1) {
    perror("fork");
```

```
    exit(EXIT_FAILURE);
}

if (cpid == 0) {    /* Child reads from pipe */
    close(pipefd[1]);          /* Close unused write end */

    while (read(pipefd[0], &buf, 1) > 0)
write(STDOUT_FILENO, &buf, 1);

    write(STDOUT_FILENO, "\n", 1);
    close(pipefd[0]);
    _exit(EXIT_SUCCESS);

} else {            /* Parent writes argv[1] to pipe */
    close(pipefd[0]);          /* Close unused read end */
    write(pipefd[1], argv[1], strlen(argv[1]));
    close(pipefd[1]);          /* Reader will see EOF */
    wait(NULL);                /* Wait for child */
    exit(EXIT_SUCCESS);
}
```

**signal** (*ANSI C signal handling*) The behavior of `signal()` varies across UNIX versions, and has also varied historically across different versions of Linux. Avoid its use: use `sigaction(2)` instead.

```
typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

`signal()` sets the disposition of the signal `signum` to `handler`, which is either `SIG_IGN`, `SIG_DFL`, or the address of a programmer-defined function (a "signal handler").

If the signal signum is delivered to the process, then one of the following happens:

- If the disposition is set to `SIG_IGN`, then the signal is ignored.
- If the disposition is set to `SIG_DFL`, then the default action associated with the signal (see `signal(7)`) occurs.
- If the disposition is set to a function, then first either the disposition is reset to `SIG_DFL`, or the signal is blocked, and then `handler` is called with argument `signum`. If invocation of the handler caused the signal to be blocked, then the signal is unblocked upon return from the handler.

The signals `SIGKILL` and `SIGSTOP` cannot be caught or ignored.

**kill** (*send signal to a process*) The `kill()` system call can be used to send any signal to any process group or process.

```
int kill(pid_t pid, int sig);
```

If `pid` is positive, then signal `sig` is sent to the process with the ID specified by `pid`.

If `pid` equals 0, then `sig` is sent to every process in the process group of the calling process.

If `pid` equals -1, then `sig` is sent to every process for which the calling process has permission to send signals, except for process 1 (`init`), but see below.

If `pid` is less than -1, then `sig` is sent to every process in the process group whose ID is `-pid`.

If `sig` is 0, then no signal is sent, but existence and permission checks are still performed; this can be used to check for the existence of a process ID or process group ID that the caller is permitted to signal.

For a process to have permission to send a signal, it must either be privileged (under Linux: have the `CAP_KILL` capability in the user namespace of the target process), or the real or effective user ID of the sending process must equal the real or saved set-user-ID of the target process. In the case of `SIGCONT`, it suffices when the sending and receiving processes belong to the same session.

**pause** (*wait for signal*) `pause()` causes the calling process (or thread) to sleep until a signal is delivered that either terminates the process or causes the invocation of a signal-catching function.

```
int pause(void);
```

`pause()` returns only when a signal was caught and the signal-catching function returned. In this case, `pause()` returns -1, and `errno` is set to `EINTR`.

**sigprocmask** (*examine and change blocked signals*) `sigprocmask()` is used to fetch and/or change the signal mask of the calling thread. The signal mask is the set of signals whose delivery is currently blocked for the caller (see also `signal(7)` for more details).

The behavior of the call is dependent on the value of `how`, as follows.

`SIG_BLOCK` The set of blocked signals is the union of the current set and the `set` argument.

`SIG_UNBLOCK` The signals in `set` are removed from the current set of blocked signals. It is permissible to attempt to unblock a signal which is not blocked.

`SIG_SETMASK` The set of blocked signals is set to the argument `set`.

If oldset is non-`NULL`, the previous value of the signal mask is stored in `oldset`.

If set is `NULL`, then the signal mask is unchanged (i.e., `how` is ignored), but the current value of the signal mask is nevertheless returned in `oldset` (if it is not `NULL`).

**sigpending** (*pending signals*) `sigpending()` returns the set of signals that are pending for delivery to the calling thread (i.e., the signals which have been raised while blocked). The mask of pending signals is returned in `set`.

```
int sigpending(sigset_t *set);
```

**sigsuspend** (*wait for a signal*) `sigsuspend()` temporarily replaces the signal mask of the calling process with the mask given by `mask` and then suspends the process until delivery of a signal whose action is to invoke a signal handler or to terminate a process.

```
int sigsuspend(const sigset_t *mask);
```

**sigaction** (*examine and change a signal action*) The `sigaction()` system call is used to change the action taken by a process on receipt of a specific signal. (See `signal(7)` for an overview of signals.)

`signum` specifies the signal and can be any valid signal except `SIGKILL` and `SIGSTOP`.

If `act` is non-`NULL`, the new action for signal `signum` is installed from `act`. If oldact is non-`NULL`, the previous action is saved in `oldact`.

The `sigaction` structure is defined as something like:

```
struct sigaction {
    void     (*sa_handler)(int);
    sigset_t   sa_mask;
    int        sa_flags;
};
```

`sa_handler` specifies the action to be associated with `signum` and may be `SIG_DFL` for the default action, `SIG_IGN` to ignore this signal, or a pointer to a signal handling function. This function receives the signal number as its only argument.

`sa_mask` specifies a mask of signals which should be blocked (i.e., added to the signal mask of the thread in which the signal handler is invoked) during execution of the signal handler. In addition, the signal which triggered the handler will be blocked, unless the `SA_NODEFER` flag is used.

`sa_flags` specifies a set of flags which modify the behavior of the signal. It is formed by the bitwise OR of zero or more of the following:

`SA_NOCLDSTOP` If signum is `SIGCHLD`, do not receive notification when child processes stop or resume.

`SA_RESETHAND` Restore the signal action to the default upon entry to the signal handler. This flag is meaningful only when establishing a signal handler. `SA_ONESHOT` is an obsolete, nonstandard synonym for this flag.

`SA_ONSTACK` Call the signal handler on an alternate signal stack provided by `sigaltstack(2)`.

`SA_NOCLDWAIT` **(since Linux 2.6)** If `signum` is `SIGCHLD`, do not transform children into zombies when they terminate.

`SA_NODEFER` Do not prevent the signal from being received from within its own signal handler.

`SA_RESTART` Provide behavior compatible with BSD signal semantics by making certain system calls restartable across signals.

**select** (*synchronous I/O multiplexing*) `select()` allow a program to monitor multiple file descriptors, waiting until one or more of the file descriptors become "ready" for some class of I/O operation (e.g., input possible). A file descriptor is considered ready if it is possible to perform a corresponding I/O operation (e.g., `read(2)` without blocking, or a sufficiently small `write(2)`).

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
   fd_set *exceptfds, struct timeval *timeout);
```

`select()` can monitor only file descriptors numbers that are less than `FD_SETSIZE`; `poll(2)` does not have this limitation.

Three independent sets of file descriptors are watched. The file descriptors listed in `readfds` will be watched to see if characters become available for reading (more precisely, to see if a read will not block; in particular, a file descriptor is also ready on end-of-file). The file descriptors in `writefds` will be watched to see if space is available for write (though a large write may still block). The file descriptors in `exceptfds` will be watched for exceptional conditions.

On exit, each of the file descriptor sets is modified in place to indicate which file descriptors actually changed status.

Four macros are provided to manipulate the sets. `FD_ZERO()` clears a set. `FD_SET()` and `FD_CLR()` respectively add and remove a given file descriptor from a set. `FD_ISSET()` tests to see if a file descriptor is part of the set; this is useful after `select()` returns.

```
void FD_CLR(int fd, fd_set *set);
int  FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);
```

`nfds` should be set to the highest-numbered file descriptor in any of the three sets, plus 1. The indicated file descriptors in each set are checked, up to this limit.

The `timeout` argument specifies the interval that `select()` should block waiting for a file descriptor to become ready.

On success, `select()` return the number of file descriptors contained in the three returned descriptor sets.

**poll** (*wait for some event on a file descriptor*) `poll()` performs a similar task to `select(2)`: it waits for one of a set of file descriptors to become ready to perform I/O.

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

The set of file descriptors to be monitored is specified in the `fds` argument, which is an array of structures of the following form:

```
struct pollfd {
    int   fd;         /* file descriptor */
    short events;     /* requested events */
    short revents;    /* returned events */
};
```

The caller should specify the number of items in the `fds` array in `nfds`.

The field `fd` contains a file descriptor for an open file.

The field `events` is an input parameter, a bit mask specifying the events the application is interested in for the file descriptor `fd`. This field may be specified as zero, in which case the only events that can be returned in `revents` are `POLLHUP`, `POLLERR`, and `POLLNVAL` (see below).

The field `revents` is an output parameter, filled by the kernel with the events that actually occurred. The bits returned in `revents` can include any of those specified in `events`, or one of the values `POLLERR`, `POLLHUP`, or `POLLNVAL`.

If none of the events requested (and no error) has occurred for any of the file descriptors, then `poll()` blocks until one of the events occurs.

The `timeout` argument specifies the number of milliseconds that `poll()` should block waiting for a file descriptor to become ready.

The bits that may be set/returned in `events` and `revents` are defined in `<poll.h>`:

`POLLIN` There is data to read.

`POLLRDNORM` Equivalent to `POLLIN`.

`POLLRDBAND` Priority band data can be read (generally unused on Linux).

`POLLPRI` There is some exceptional condition on the file descriptor.

`POLLOUT` Writing is now possible, though a write larger that the available space in a socket or pipe will still block (unless `O_NONBLOCK` is set).

`POLLWRNORM` Equivalent to `POLLOUT`.

`POLLWRBAND` Priority data may be written.

These three bits are meaningless in the `events` field, and will be set in the `revents` field whenever the corresponding condition is true:

`POLLERR` Error condition (only returned in `revents`; ignored in `events`). This bit is also set for a file descriptor referring to the write end of a pipe when the read end has been closed.

`POLLHUP` Hang up (only returned in `revents`; ignored in `events`). Note that when reading from a channel such as a pipe or a stream socket, this event merely indicates that the peer closed its end of the channel. Subsequent reads from the channel will return 0 (end of file) only after all outstanding data in the channel has been consumed.

`POLLNVAL` Invalid request: `fd` not open (only returned in `revents`; ignored in `events`).

## 10.2 Library calls (3)

**raise** (*send a signal to the caller*)

```
int raise(int sig);
```

The `raise()` function sends a signal to the calling process or thread. In a single-threaded program it is equivalent to

```
kill(getpid(), sig);
```

In a multithreaded program it is equivalent to

```
pthread_kill(pthread_self(), sig);
```

If the signal causes a handler to be called, `raise()` will return only after the signal handler has returned.

**sigemptyset, sigfillset, sigaddset, sigdelset, sigismember** (*POSIX signal set operations*) These functions allow the manipulation of POSIX signal sets.

```
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);
```

`sigemptyset()` initializes the signal `set` given by set to empty, with all signals excluded from the set.

`sigfillset()` initializes `set` to full, including all signals.

`sigaddset()` and `sigdelset()` add and delete respectively signal `signum` from `set`.

`sigismember()` tests whether `signum` is a member of `set`.

Objects of type `sigset_t` must be initialized by a call to either `sigemptyset()` or `sigfillset()` before being passed to the functions `sigaddset()`, `sigdelset()` and `sigismember()` or the additional glibc functions described below (`sigisemptyset()`, `sigandset()`, and `sigorset()`). The results are undefined if this is not done.

## 10.3 Miscellaneous (7)

**pipe** (*overview of pipes and FIFOs*) Pipes and FIFOs (also known as named pipes) provide a unidirectional interprocess communication channel. A pipe has a read end and a write end. Data written to the write end of a pipe can be read from the read end of the pipe.

A pipe is created using `pipe(2)`, which creates a new pipe and returns two file descriptors, one referring to the read end of the pipe, the other referring to the write end. Pipes can be used to create a communication channel between related processes; see `pipe(2)` for an example.

`PIPE_BUF` POSIX.1 says that `write(2)`-s of less than `PIPE_BUF` bytes must be atomic: the output data is written to the pipe as a contiguous sequence. Writes of more than `PIPE_BUF` bytes may be nonatomic: the kernel may interleave the data with data written by other processes.

**signal** (*overview of signals*) Linux supports both POSIX reliable signals (hereinafter "standard signals") and POSIX real-time signals.

| Signal | Value | Action | Comment |
|---|---|---|---|
| SIGABRT | 6 | Core | Abort signal from abort(3) |
| SIGALRM | 14 | Term | Timer signal from alarm(2) |
| SIGBUS | 10,7,10 | Core | Bus error (bad memory access) |
| SIGCHLD | 20,17,18 | Ign | Child stopped or terminated |
| SIGCONT | 19,18,25 | Cont | Continue if stopped |
| SIGFPE | 8 | Core | Floating-point exception |
| SIGHUP | 1 | Term | Hangup detected on controlling terminal or death of controlling process |
| SIGILL | 4 | Core | Illegal Instruction |
| SIGINT | 2 | Term | Interrupt from keyboard |
| SIGKILL | 9 | Term | Kill signal |
| SIGPIPE | 13 | Term | Broken pipe: write to pipe with no readers; see pipe(7) |
| SIGQUIT | 3 | Core | Quit from keyboard |
| SIGSEGV | 11 | Core | Invalid memory reference |
| SIGSTOP | 17,19,23 | Stop | Stop process |
| SIGTERM | 15 | Term | Termination signal |
| SIGTSTP | 18,20,24 | Stop | Stop typed at terminal |
| SIGTTIN | 21,21,26 | Stop | Terminal input for background process |
| SIGTTOU | 22,22,27 | Stop | Terminal output for background process |
| SIGUSR1 | 30,10,16 | Term | User-defined signal 1 |
| SIGUSR2 | 31,12,17 | Term | User-defined signal 2 |

# 11 Advanced I/O operations

## 11.1 System calls (2)

**readv, writev** (*read or write data into multiple buffers*) The `readv()` system call reads `iovcnt` buffers from the file associated with the file descriptor `fd` into the buffers described by `iov` ("scatter input").

```
ssize_t readv(int fd, const struct iovec *iov, int iovcnt);
```

The `writev()` system call writes `iovcnt` buffers of data described by `iov` to the file associated with the file descriptor `fd` ("gather output").

```
ssize_t writev(int fd, const struct iovec *iov, int iovcnt);
```

The pointer `iov` points to an array of `iovec` structures, defined in `<sys/uio.h>` as:

```
struct iovec {
    void  *iov_base;    /* Starting address */
    size_t iov_len;     /* Number of bytes to transfer */
};
```

The `readv()` system call works just like `read(2)` except that multiple buffers are filled.

The `writev()` system call works just like `write(2)` except that multiple buffers are written out.

Buffers are processed in array order. This means that `readv()` completely fills `iov[0]` before proceeding to `iov[1]`, and so on. Similarly, `writev()` writes out the entire contents of `iov[0]` before proceeding to `iov[1]`, and so on.

The data transfers performed by `readv()` and `writev()` are atomic.

On success, `readv()` return the number of bytes read; `writev()` return the number of bytes written.

**mmap** (*map files or devices into memory*) `mmap()` creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in `addr`. The `length` argument specifies the length of the mapping (which must be greater than 0).

```
void *mmap(void *addr, size_t length, int prot, int flags,
   int fd, off_t offset);
```

If `addr` is `NULL`, then the kernel chooses the (page-aligned) address at which to create the mapping; this is the most portable method of creating a new mapping. If `addr` is not `NULL`, then the kernel takes it as a hint about where to place the mapping; on Linux, the mapping will be created at a nearby page boundary.

The contents of a file mapping, are initialized using `length` bytes starting at offset `offset` in the file (or other object) referred to by the file descriptor `fd`.

The `prot` argument describes the desired memory protection of the mapping (and must not conflict with the open mode of the file). It is either `PROT_NONE` or the bitwise OR of one or more of the following flags:

`PROT_EXEC` Pages may be executed.

`PROT_READ` Pages may be read.

`PROT_WRITE` Pages may be written.

`PROT_NONE` Pages may not be accessed.

The flags argument determines whether updates to the mapping are visible to other processes mapping the same region, and whether updates are carried through to the underlying file. This behavior is determined by including exactly one of the following values in flags:

`MAP_SHARED` Share this mapping. Updates to the mapping are visible to other processes mapping the same region, and (in the case of file-backed mappings) are carried through to the underlying file. (To precisely control when updates are carried through to the underlying file requires the use of `msync(2)`.)

`MAP_PRIVATE` Create a private copy-on-write mapping. Updates to the mapping are not visible to other processes mapping the same file, and are not carried through to the underlying file. It is unspecified whether changes made to the file after the `mmap()` call are visible in the mapped region.

In addition, zero or more of the following values can be ORed in `flags`:

`MAP_FIXED` Don't interpret `addr` as a hint: place the mapping at exactly that address. `addr` must be suitably aligned: for most architectures a multiple of the page size is sufficient; however, some architectures may impose additional restrictions. If the memory region specified by `addr` and `len` overlaps pages of any existing mapping(s), then the overlapped part of the existing mapping(s) will be discarded. If the specified address cannot be used, `mmap()` will fail.

On success, `mmap()` returns a pointer to the mapped area.

**munmap** (*unmap files or devices into memory*) The `munmap()` system call deletes the mappings for the specified address range, and causes further references to addresses within the range to generate invalid memory references. The region is also automatically unmapped when the process is terminated. On the other hand, closing the file descriptor does not unmap the region.

```
int munmap(void *addr, size_t length);
```

The address `addr` must be a multiple of the page size (but `length` need not be). All pages containing a part of the indicated range are unmapped, and subsequent references to these pages will generate `SIGSEGV`. It is not an error if the indicated range does not contain any mapped pages.

**msync** (*synchronize a file with a memory map*) `msync()` flushes changes made to the in-core copy of a file that was mapped into memory using `mmap(2)` back to the filesystem. Without use of this call, there is no guarantee that changes are written back before `munmap(2)` is called. To be more precise, the part of the file that corresponds to the memory area starting at `addr` and having length `length` is updated.

```
int msync(void *addr, size_t length, int flags);
```

The `flags` argument should specify exactly one of `MS_ASYNC` and `MS_SYNC`, and may additionally include the `MS_INVALIDATE` bit. These bits have the following meanings:

`MS_ASYNC` Specifies that an update be scheduled, but the call returns immediately.

`MS_SYNC` Requests an update and waits for it to complete.

`MS_INVALIDATE` Asks to invalidate other mappings of the same file (so that they can be updated with the fresh values just written).

**mprotect** (*set protection on a region of memory*) `mprotect()` changes the access protections for the calling process's memory pages containing any part of the address range in the interval `[addr, addr+len-1]`. `addr` must be aligned to a page boundary.

```
int mprotect(void *addr, size_t len,
     int prot); // prot is same as in the mmap
```

**mlock, mlock2, munlock, mlockall, munlockall** (*lock and unlock memory*) `mlock()`, `mlock2()` and `mlockall()` lock part or all of the calling process's virtual address space into RAM, preventing that memory from being paged to the swap area.

```
int mlock(const void *addr, size_t len);
int mlock2(const void *addr, size_t len, int flags);
int mlockall(int flags);
```

`munlock()` and `munlockall()` perform the converse operation, unlocking part or all of the calling process's virtual address space, so that pages in the specified virtual address range may once more to be swapped out if required by the kernel memory manager.

```
int munlock(const void *addr, size_t len);
int munlockall(void);
```

The `flags` argument is constructed as the bitwise OR of one or more of the following constants:

`MCL_CURRENT` Lock all pages which are currently mapped into the address space of the process.

`MCL_FUTURE` Lock all pages which will become mapped into the address space of the process in the future. These could be, for instance, new pages required by a growing heap and stack as well as new memory-mapped files or shared memory regions.

# 12 Threads

## 12.1 Library calls (3)

**pthread_create** (*create a new thread*) The `pthread_create()` function starts a new thread in the calling process. The new thread starts execution by invoking `start_routine()`; `arg` is passed as the sole argument of `start_routine()`.

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
   void *(*start_routine) (void *), void *arg);
```

The `attr` argument points to a `pthread_attr_t` structure whose contents are used at thread creation time to determine attributes for the new thread; this structure is initialized using `pthread_attr_init(3)` and related functions. If `attr` is `NULL`, then the thread is created with default attributes.

Before returning, a successful call to `pthread_create()` stores the ID of the new thread in the buffer pointed to by `thread`; this identifier is used to refer to the thread in subsequent calls to other pthreads functions.

By default, a new thread is created in a joinable state, unless `attr` was set to create the thread in a detached state (using `pthread_attr_setdetachstate(3)`).

**pthread_exit** (*terminate calling thread*) The `pthread_exit()` function terminates the calling thread and returns a value via `retval` that (if the thread is joinable) is available to another thread in the same process that calls `pthread_join(3)`.

Any clean-up handlers established by `pthread_cleanup_push(3)` that have not yet been popped, are popped (in the reverse of the order in which they were pushed) and executed. If the thread has any thread-specific data, then, after the clean-up handlers have been executed, the corresponding destructor functions are called, in an unspecified order.

When a thread terminates, process-shared resources (e.g., mutexes, condition variables, semaphores, and file descriptors) are not released, and functions registered using `atexit(3)` are not called.

After the last thread in a process terminates, the process terminates as by calling `exit(3)` with an exit status of zero; thus, process-shared resources are released and functions registered using `atexit(3)` are called.

**pthread_join** (*join with a terminated thread*) The `pthread_join()` function waits for the thread specified by `thread` to terminate. If that thread has already terminated, then `pthread_join()` returns immediately. The thread specified by `thread` must be joinable.

```
int pthread_join(pthread_t thread, void **retval);
```

If `retval` is not `NULL`, then `pthread_join()` copies the exit status of the target thread (i.e., the value that the target thread supplied to `pthread_exit(3)`) into the location pointed to by `retval`.

**pthread_mutex_destroy, pthread_mutex_init** (*destroy and initialize a mutex*) The `pthread_mutex_destroy()` function shall destroy the mutex object referenced by `mutex`; the mutex object becomes, in effect, uninitialized. An implementation may cause `pthread_mutex_destroy()` to set the object referenced by `mutex` to an invalid value.

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

A destroyed mutex object can be reinitialized using `pthread_mutex_init()`; the results of otherwise referencing the object after it has been destroyed are undefined.

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
    const pthread_mutexattr_t *restrict attr);
```

The `pthread_mutex_init()` function shall initialize the mutex referenced by `mutex` with attributes specified by `attr`. If `attr` is `NULL`, the default mutex attributes are used; the effect shall be the same as passing the address of a default mutex attributes object. Upon successful initialization, the state of the mutex becomes initialized and unlocked.

**pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock** (*lock and unlock a mutex*) The mutex object referenced by `mutex` shall be locked by a call to `pthread_mutex_lock()` that returns zero or `[EOWNERDEAD]`. If the mutex is already locked by another thread, the calling thread shall block until the mutex becomes available. This operation shall return with the mutex object referenced by `mutex` in the locked state with the calling thread as its owner.

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

The `pthread_mutex_trylock()` function shall be equivalent to `pthread_mutex_lock()`, except that if the mutex object referenced by `mutex` is currently locked (by any thread, including the current thread), the call shall return immediately.

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

The `pthread_mutex_unlock()` function shall release the mutex object referenced by `mutex`. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by `mutex` when `pthread_mutex_unlock()` is called, resulting in the mutex becoming available, the scheduling policy shall determine which thread shall acquire the mutex.

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Mutex objects are intended to serve as a low-level primitive from which other thread synchronization functions can be built. As such, the implementation of mutexes should be as efficient as possible, and this has ramifications on the features available at the interface.

**pthread_cond_timedwait, pthread_cond_wait** (*wait on a condition*) The `pthread_cond_timedwait()` and `pthread_cond_wait()` functions shall block on a condition variable. The application shall ensure that these functions are called with `mutex` locked by the calling thread.

```
int pthread_cond_timedwait(pthread_cond_t *restrict cond,
    pthread_mutex_t *restrict mutex,
    const struct timespec *restrict abstime);

int pthread_cond_wait(pthread_cond_t *restrict cond,
    pthread_mutex_t *restrict mutex);
```

When using condition variables there is always a Boolean predicate involving shared variables associated with each condition wait that is true if the thread should proceed.

**pthread_cond_broadcast, pthread_cond_signal** (*broadcast or signal a condition*) These functions shall unblock threads blocked on a condition variable.

The `pthread_cond_broadcast()` function shall unblock all threads currently blocked on the spec- ified condition variable `cond`.

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

The `pthread_cond_signal()` function shall unblock at least one of the threads that are blocked on the specified condition variable `cond` (if any threads are blocked on `cond`).

```
int pthread_cond_signal(pthread_cond_t *cond);
```

**pthread_cond_destroy** (*destroy condition variables*) The `pthread_cond_destroy()` function shall destroy the given condition variable specified by `cond`; the object becomes, in effect, uninitialized. An implementation may cause `pthread_cond_destroy()` to set the object referenced by `cond` to an invalid value. A destroyed condition variable object can be reinitialized using `pthread_cond_init()`; the results of otherwise referencing the object after it has been destroyed are undefined.

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

**pthread_key_create** (*thread-specific data key creation*) The `pthread_key_create()` function shall create a thread-specific data key visible to all threads in the process. Key values provided by `pthread_key_create()` are opaque objects used to locate thread-specific data. Although the same key value may be used by different threads, the values bound to the key by `pthread_setspecific()` are maintained on a per-thread basis and persist for the life of the calling thread.

```
int pthread_key_create(pthread_key_t *key, void (*destructor)(void*));
```

**pthread_getspecific, pthread_setspecific** (*thread-specific data management*) The `pthread_getspecific()` function shall return the value currently bound to the specified `key` on behalf of the calling thread.

```
void *pthread_getspecific(pthread_key_t key);
```

The `pthread_setspecific()` function shall associate a thread-specific `value` with a `key` obtained via a previous call to `pthread_key_create()`. Different threads may bind different values to the same key. These values are typically pointers to blocks of dynamically allocated memory that have been reserved for use by the calling thread.

```
int pthread_setspecific(pthread_key_t key, const void *value);
```

**pthread_key_delete** (*thread-specific data key deletion*) The `pthread_key_delete()` function shall delete a thread-specific data key previously returned by `pthread_key_create()`. The thread-specific data values associated with `key` need not be `NULL` at the time `pthread_key_delete()` is called. It is the responsibility of the application to free any application storage or perform any cleanup actions for data structures related to the deleted key or associated thread-specific data in any threads; this cleanup can be done either before or after `pthread_key_delete()` is called. Any attempt to use `key` following the call to `pthread_key_delete()` results in undefined behavior.

```
int pthread_key_delete(pthread_key_t key);
```

## 12.2 Miscellaneous (7)

**pthreads** (*POSIX threads*) POSIX.1 specifies a set of interfaces (functions, header files) for threaded programming commonly known as POSIX threads, or Pthreads. A single process can contain multiple threads, all of which are executing the same program. These threads share the same global memory (data and heap segments), but each thread has its own stack (automatic variables).

**Linux implementations of POSIX threads** Over time, two threading implementations have been provided by the GNU C library on Linux:

**LinuxThreads** This is the original Pthreads implementation. Since glibc 2.4, this implementation is no longer supported.

**NPTL (Native POSIX Threads Library)** This is the modern Pthreads implementation. By comparison with LinuxThreads, NPTL provides closer conformance to the requirements of the POSIX.1 specification and better performance when creating large numbers of threads. NPTL is available since glibc 2.3.2, and requires features that are present in the Linux 2.6 kernel.

Both of these are so-called 1:1 implementations, meaning that each thread maps to a kernel scheduling entity. Both threading implementations employ the Linux `clone(2)` system call. In NPTL, thread synchronization primitives (mutexes, thread joining, and so on) are implemented using the Linux `futex(2)` system call.