POSIX system calls for regular files and processes

What is POSIX?

POSIX (pronounced /ˈpɒzɪks/) or "Portable Operating System Interface for Unix" is the name of a family of related standards specified by the IEEE to define the application programming interface (API), along with shell and utilities interfaces for software compatible with variants of the Unix operating system, although the standard can apply to any operating system.

What is a system call?

A system call is the mechanism used by an application program **to request service from the operating system** based on the monolithic kernel or to system servers on operating systems based on the microkernel-structure.
A system call is a request made by any program to the operating system for performing tasks -- **picked from a predefined set** -- which the said program does not have required permissions to execute in its own flow of execution.
System calls provide the interface between a process and the operating system.

System calls for regular files

- creat
- open
- close
- read
- write
- lseek

```
#include <fcntl.h>

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

DESCRIPTION

The function call:

creat(path, mode)

shall be equivalent to:

open(path, O_WRONLY|O_CREAT|O_TRUNC, mode)

RETURN VALUE

Refer to open().

```
#include <fcntl.h>

int open(const char *path, int oflag, ...);
```

DESCRIPTION

The open() function shall establish the connection between a file and a file descriptor. It shall create an open file description that refers to a file and a file descriptor that refers to that open file description. The file descriptor is used by other I/O functions to refer to that file. The path argument points to a pathname naming the file.

The open() function shall return a file descriptor for the named file that is the lowest file descriptor not currently open for that process. The open file description is new, and therefore the file descriptor shall not share it with any other process in the system.

The file offset used to mark the current position within the file shall be set to the beginning of the file.

```
#include <fcntl.h>

int open(const char *path, int oflag, ...);
```

DESCRIPTION (2)

The file status flags and file access modes of the open file description shall be set according to the value of oflag.

Values for oflag are constructed by a bitwise-inclusive OR of flags from the following list, defined in <fcntl.h>.

Applications shall specify **exactly one** of the first three values (file access modes) below in the value of oflag:

O_RDONLY
Open for reading only.
O_WRONLY
Open for writing only.
O_RDWR
Open for reading and writing. The result is undefined if this flag is applied to a FIFO.

```
#include <fcntl.h>

int open(const char *path, int oflag, ...);
```

DESCRIPTION (3)

Any combination of the following may be used:

O_CREAT
If the file exists, this flag has no effect except as noted under O_EXCL below. Otherwise, the file shall be created and the access permission bits of the file mode shall be set to the value of the **third** argument taken as type mode_t. When bits other than the file permission bits are set, the effect is unspecified. The third argument does not affect whether the file is open for reading, writing, or for both.

O_EXCL
If O_CREAT and O_EXCL are set, open() shall fail if the file exists. If O_EXCL is set and O_CREAT is not set, the result is undefined.

```
#include <fcntl.h>

int open(const char *path, int oflag, ...);
```

DESCRIPTION (4)

O_APPEND
If set, the file offset shall be set to the end of the file prior to each write.

O_TRUNC
If the file exists and is a regular file, and the file is successfully opened O_RDWR or O_WRONLY, its length shall be truncated to 0, and the mode and owner shall be unchanged. It shall have no effect on FIFO special files or terminal device files. Its effect on other file types is implementation-defined. The result of using O_TRUNC with O_RDONLY is undefined.

```
#include <fcntl.h>

int open(const char *path, int oflag, ...);
```

RETURN VALUE

Upon successful completion, the function shall open the file and return a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, -1 shall be returned and errno set to indicate the error. **No files shall be created or modified if the function returns -1.**

```
#include <unistd.h>

int close(int fildes);
```

DESCRIPTION

The close() function shall deallocate the file descriptor indicated by fildes. To deallocate means to make the file descriptor available for return by subsequent calls to open() or other functions that allocate file descriptors. All outstanding record locks owned by the process on the file associated with the file descriptor shall be removed (that is, unlocked).

RETURN VALUE

Upon successful completion, 0 shall be returned; otherwise, -1 shall be returned and errno set to indicate the error.

```c
#include <unistd.h>

ssize_t read(int fildes, void *buf,
size_t nbyte);
```

DESCRIPTION

The read() function shall attempt to read nbyte bytes from the file associated with the open file descriptor, fildes, into the buffer pointed to by buf. The behavior of multiple concurrent reads on the same pipe, FIFO, or terminal device is unspecified.

Before any action described below is taken, and if nbyte is zero, the read() function may detect and return errors as described below. In the absence of errors, or if error detection is not performed, the read() function shall return zero and have no other results.

```
#include <unistd.h>

ssize_t read(int fildes, void *buf,
size_t nbyte);
```

DESCRIPTION (2)

On files that support seeking (for example, a regular file), the read() shall start at a position in the file given by the file offset associated with fildes. The file offset shall be incremented by the number of bytes actually read.

Files that do not support seeking-for example, terminals-always read from the current position. The value of a file offset associated with such a file is undefined.

No data transfer shall occur past the current end-of-file. If the starting position is at or after the end-of-file, 0 shall be returned.

If the value of nbyte is greater than {SSIZE_MAX}, the result is implementation-defined.

```
#include <unistd.h>

ssize_t read(int fildes, void *buf,
size_t nbyte);
```

DESCRIPTION (3)

The read() function reads data previously written to a file. If any portion of a regular file prior to the end-of-file has not been written, read() shall return bytes with value 0. For example, lseek() allows the file offset to be set beyond the end of existing data in the file. If data is later written at this point, subsequent reads in the gap between the previous end of data and the newly written data shall return bytes with value 0 until data is written into the gap.

RETURN VALUE

Upon successful completion, read() shall return a non-negative integer indicating the number of bytes actually read. Otherwise, the function shall return -1 and set errno to indicate the error.

```
#include <unistd.h>

ssize_t write(int fildes, const void
*buf, size_t nbyte);
```

DESCRIPTION

The write() function shall attempt to write nbyte bytes from the buffer pointed to by buf to the file associated with the open file descriptor, fildes.

Before any action described below is taken, and if nbyte is zero and the file is a regular file, the write() function may detect and return errors as described below. In the absence of errors, or if error detection is not performed, the write() function shall return zero and have no other results. If nbyte is zero and the file is not a regular file, the results are unspecified.

```
#include <unistd.h>

ssize_t write(int fildes, const void
*buf, size_t nbyte);
```

DESCRIPTION (2)

On a regular file or other file capable of seeking, the actual writing of data shall proceed from the position in the file indicated by the file offset associated with fildes. Before successful return from write(), the file offset shall be incremented by the number of bytes actually written. On a regular file, if this incremented file offset is greater than the length of the file, the length of the file shall be set to this file offset.

On a file not capable of seeking, writing shall always take place starting at the current position. The value of a file offset associated with such a device is undefined.

```
#include <unistd.h>

ssize_t write(int fildes, const void
*buf, size_t nbyte);
```

DESCRIPTION (3)

If the O_APPEND flag of the file status flags is set, the file offset shall be set to the end of the file prior to each write and no intervening file modification operation shall occur between changing the file offset and the write operation.

If a write() requests that more bytes be written than there is room for, only as many bytes as there is room for shall be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512 bytes will return 20. The next write of a non-zero number of bytes would give a failure return (except as noted below).

```
#include <unistd.h>

ssize_t write(int fildes, const void
*buf, size_t nbyte);
```

DESCRIPTION (4)

After a write() to a regular file has successfully returned:

- Any successful read() from each byte position in the file that was modified by that write shall return the data specified by the write() for that position until such byte positions are again modified.
- Any subsequent successful write() to the same byte position in the file shall overwrite that file data.

RETURN VALUE

Upon successful completion, write() shall return the number of bytes actually written to the file associated with fildes. This number shall never be greater than nbyte. Otherwise, -1 shall be returned and errno set to indicate the error.

```
#include <unistd.h>

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

DESCRIPTION

The lseek() function shall set the file offset for the open file description associated with the file descriptor fildes, as follows:

- If whence is SEEK_SET, the file offset shall be set to offset bytes.
- If whence is SEEK_CUR, the file offset shall be set to its current location plus offset.
- If whence is SEEK_END, the file offset shall be set to the size of the file plus offset.

The symbolic constants SEEK_SET, SEEK_CUR, and SEEK_END are defined in <unistd.h>.

```
#include <unistd.h>

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

DESCRIPTION (2)

The behavior of lseek() on devices which are incapable of seeking is implementation-defined. The value of the file offset associated with such a device is undefined.

The lseek() function shall allow the file offset to be set beyond the end of the existing data in the file. If data is later written at this point, subsequent reads of data in the gap shall return bytes with the value 0 until data is actually written into the gap.

The lseek() function shall not, by itself, extend the size of a file.

RETURN VALUE

Upon successful completion, the resulting offset, as measured in bytes from the beginning of the file, shall be returned. Otherwise, (off_t)-1 shall be returned, errno shall be set to indicate the error, and the file offset shall remain unchanged.

System calls for processes

- exit
- fork
- exec*
- wait, waitpid
- getpid, getppid

```c
#include <stdlib.h>

void exit(int status);
```

DESCRIPTION

The value of status may be 0, EXIT_SUCCESS, EXIT_FAILURE, or any other value, though only the least significant 8 bits (that is, status & 0377) shall be available to a waiting parent process.

The exit() function shall first call all functions registered by atexit(), in the reverse order of their registration,

The exit() function shall then flush all open streams with unwritten buffered data, close all open streams, and remove all files created by tmpfile().

Finally, exit() calls _exit() which shall terminate the calling process. All of the file descriptors open in the calling process shall be closed.

If the parent process of the calling process is not executing a wait() or waitpid(), the calling process shall be transformed into a zombie process. A zombie process is an inactive process and it shall be deleted at some later time when its parent process executes wait() or waitpid().

```c
#include <stdlib.h>

void exit(int status);
```

DESCRIPTION (2)

Termination of a process does not directly terminate its children. The sending of a SIGHUP signal indirectly terminates children in **some** circumstances.

The parent process ID of all of the calling process' existing child processes and zombie processes shall be set to the process ID of an implementation-defined system process. That is, these processes shall be inherited by a special system process.

RETURN VALUE

These functions do not return.

ERRORS

No errors are defined.

```
#include <unistd.h>

pid_t fork(void);
```

DESCRIPTION

The fork() function shall create a new process. The new process (child process) shall be an exact copy of the calling process (parent process) except:

- The child process shall have a unique process ID.
- The child process ID also shall not match any active process group ID.
- The child process shall have a different parent process ID, which shall be the process ID of the calling process.
- The child process shall have its own copy of the parent's file descriptors. **Each of the child's file descriptors shall refer to the same open file description with the corresponding file descriptor of the parent.**
- The child process' values of tms_utime, tms_stime, tms_cutime, and tms_cstime shall be set to 0.
- The initial value of the CPU-time clock of the child process shall be set to zero.
- The initial value of the CPU-time clock of the single thread of the child process shall be set to zero.
- *others*

After fork(), both the parent and the child processes shall be capable of executing independently before either one terminates.

```
#include <unistd.h>

pid_t fork(void);
```

RETURN VALUE

Upon successful completion, fork() shall return 0 to the child process and shall return the process ID of the child process to the parent process. Both processes shall continue to execute from the fork() function. Otherwise, -1 shall be returned to the parent process, no child process shall be created, and errno shall be set to indicate the error.

```
#include <unistd.h>

int execl(const char *path, const char *arg0, ... /*, (char *)0 */);
int execv(const char *path, char *const argv[]);
int execlp(const char *file, const char *arg0, ... /*, (char *)0 */);
int execvp(const char *file, char *const argv[]);
```

DESCRIPTION

The exec family of functions shall replace the current process image with a new process image. The new image shall be constructed from a regular, executable file called the new process image file. There shall be no return from a successful exec, because the calling process image is overlaid by the new process image.

When a C-language program is executed as a result of this call, it shall be entered as a C-language function call as follows:

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

where argc is the argument count and argv is an array of character pointers to the arguments themselves.

The arguments specified by a program with one of the exec functions shall be passed on to the new process image in the corresponding main() arguments.

The argument path points to a pathname that identifies the new process image file.

The argument file is used to construct a pathname that identifies the new process image file. If the file argument contains a slash character, the file argument shall be used as the pathname for this file. Otherwise, the path prefix for this file is obtained by a search of the directories passed as the environment variable PATH.

```
#include <unistd.h>

int execl(const char *path, const char *arg0, ... /*, (char *)0 */);
int execv(const char *path, char *const argv[]);
int execlp(const char *file, const char *arg0, ... /*, (char *)0 */);
int execvp(const char *file, char *const argv[]);
```

DESCRIPTION (2)

The arguments represented by arg0,... are pointers to null-terminated character strings. These strings shall constitute the argument list available to the new process image. The list is terminated by a null pointer. The argument arg0 should point to a filename that is associated with the process being started by one of the exec functions.

The argument argv is an array of character pointers to null-terminated strings. The application shall ensure that the last member of this array is a null pointer. These strings shall constitute the argument list available to the new process image. The value in argv[0] should point to a filename that is associated with the process being started by one of the exec functions.

File descriptors open in the calling process image shall remain open in the new process image, except for those whose close-on- exec flag FD_CLOEXEC is set. For those file descriptors that remain open, all attributes of the open file description remain unchanged.

RETURN VALUE

If one of the exec functions returns to the calling process image, an error has occurred; the return value shall be -1, and errno shall be set to indicate the error.

```
#include <sys/wait.h>

pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

DESCRIPTION

The wait() and waitpid() functions shall obtain status information pertaining to one of the caller's child processes. Various options permit status information to be obtained for child processes that have terminated or stopped. If status information is available for two or more child processes, the order in which their status is reported is unspecified.

The wait() function shall suspend execution of the calling thread until status information for one of the terminated child processes of the calling process is available, or until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process. If more than one thread is suspended in wait() or waitpid() awaiting termination of the same process, exactly one thread shall return the process status at the time of the target process termination. If status information is available prior to the call to wait(), return shall be immediate.

```
#include <sys/wait.h>

pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

DESCRIPTION (2)

The waitpid() function shall be equivalent to wait() if the pid argument is (pid_t)-1 and the options argument is 0. Otherwise, its behavior shall be modified by the values of the pid and options arguments.

The pid argument specifies a set of child processes for which status is requested. The waitpid() function shall only return the status of a child process from this set:

- If pid is equal to (pid_t)-1, status is requested for any child process. In this respect, waitpid() is then equivalent to wait().
- If pid is greater than 0, it specifies the process ID of a single child process for which status is requested.
- If pid is 0, status is requested for any child process whose process group ID is equal to that of the calling process.

•If pid is less than (pid_t)-1, status is requested for any child process whose process group ID is equal to the absolute value of pid.

```
#include <sys/wait.h>

pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

DESCRIPTION (3)

If wait() or waitpid() return because the status of a child process is available, these functions shall return a value equal to the process ID of the child process. In this case, if the value of the argument stat_loc is not a null pointer, information shall be stored in the location pointed to by stat_loc. The value stored at the location pointed to by stat_loc shall be 0 if and only if the status returned is from a terminated child process that terminated by one of the following means:

1. The process returned 0 from main().
2. The process called _exit() or exit() with a status argument of 0.
3. The process was terminated because the last thread in the process terminated.

RETURN VALUE

If wait() or waitpid() returns because the status of a child process is available, these functions shall return a value equal to the process ID of the child process for which status is reported.

```
#include <unistd.h>
pid_t getpid(void);
```

DESCRIPTION

The getpid() function shall return the process ID of the calling process.

RETURN VALUE

The getpid() function shall always be successful and no return value is reserved to indicate an error.

ERRORS

No errors are defined.

```
#include <unistd.h>
pid_t getpid(void);
```

DESCRIPTION

The getppid() function shall return the parent process ID of the calling process.

RETURN VALUE

The getppid() function shall always be successful and no return value is reserved to indicate an error.

ERRORS

No errors are defined.

THE END

See also:
http://en.wikipedia.org/wiki/POSIX
http://en.wikipedia.org/wiki/System_call
POSIX headers
POSIX System Interfaces